A MULTICOMPUTER APPROACH TO NON-NUMERIC COMPUTATION

BY

CHAITANYA K. BARU

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1985

TO MY PARENTS
SESHU AND VITHAL

ACKNOWLEDGEMENTS

I wish to express my deepest and most sincere thanks to the three individuals to whom I owe everything--my mother; my father; and my advisor, Dr. Stanley Su.

I am very grateful to Dr. Sham Navathe for his help and guidance during my stay in Gainesville. I also thank my other committee members, Dr. Y. C. Chow, Dr. H. Lam, and Dr. J. Staudhammer for their participation in my committee.

My stay in Gainesville was extremely pleasant and very memorable thanks to a long list of very dear friends--Ashok, Ravi, Vishu, Liley, Kumar, Shiv, Indira and Ganesh, Rangaraj and Ranga, Sharon, and the gang at the Database Center.

Lastly, I thank Nitty, Murali, Sanju, and Rama who have been a constant source of encouragement from half-way across the world.

This research, including my assistantship, was supported by the Department of Energy contract DE-FG05-84ER13285.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A MULTICOMPUTER APPROACH TO NON-NUMERIC COMPUTATION

By

CHAITANYA K. BARU

December 1985

Chairman : Stanley Y. W. Su
Major Department: Electrical Engineering

In this research we study the architecture of a dynamically partitionable multicomputer system with switchable main memory modules (SM3) for non-numeric computations. The architecture supports efficient execution of parallel algorithms for database processing by i) allowing the sharing of switchable main memory modules between computers, ii) supporting network partitioning, and iii) employing global control lines to efficiently support inter-processor communication. The data transfer time is reduced to memory switching time by allowing some main memory modules to be switched between processors. Network partitioning gives a common bus network system the capability of an MIMD machine while performing global operations. The global control lines establish a quick and efficient high-level protocol in the system. The network is supervised by a control computer which oversees network partitioning and other global functions.

A simulation study of some commonly used database operations, using discrete-event simulation techniques, has been carried out and the results of the study are presented. Certain aspects of the system architecture have been modified on the basis of simulation results. The general-purpose nature of the SM3 system allows the implementation of a variety of parallel algorithms for database processing. We present a methodology for representing and transforming such algorithms in order to obtain the most efficient mapping of algorithms onto the underlying architecture. The methodology suggests a stepwise decomposition of the algorithm in order to exploit "vertical" and "horizontal" parallelism. The extent of vertical parallelism is determined by the structure of the algorithm whereas the extent of horizontal parallelism is determined by the distribution of data. In the transformation process, special attention is paid to the data movement and transformation aspects of the algorithm.

CHAPTER 1
INTRODUCTION

There is an ever increasing demand on the processing capability of computer systems. Research in computer architecture is driven by the fact that the rate of increase in this demand is much more than the corresponding rate of increase in raw hardware capability. The projected future demand for computing power far exceeds the anticipated development in the traditional hardware technologies. In other words, the rate of increase of, say switching times, circuit densities, etc., is not nearly enough to match users' demands. Thus, the inherent limitations of the underlying technology automatically place a limitation on the processing capability of a uniprocessor system. The restrictions of uniprocessor systems become even more apparent in the context of database management systems where the rate of increase in size (of the database) and complexity (of queries) is again very rapid. In addition, users expect system response times to decrease constantly with progress in technology, even as database processing becomes more complex.

The effort to build computer systems which satisfy the users' demands for database processing led to the development of database machines. Many database machines have been proposed in the past for efficient processing of databases [BAN79, BAT77, CAN74, DEW79, IEE79a, IEE79b, LIN76, OZK75, SCH79, SU79, SU83]. A few such machines like IDM-500 [EPS80], iDBP, and Terradata [IEE84] are even available commercially. Much of the past work in database machines has been aimed at designing and constructing special purpose systems using dedicated hardware to support database processing.

1.1 A Classification of Database Machines

Database machines can be categorized according to the architectural techniques used to support efficient database processing. The architecture of a "conventional" or "von Neumann" computer is shown in Figure 1.1. It consists of three distinct components--secondary store (I/O), primary memory (PM), and a central processing unit (CPU). The commands for processing data are executed by the CPU. The data need to be staged from I/O to PM and from PM to CPU before they can be processed. The need to stage data from I/O to PM gives rise to an overhead which is referred to as the "I/O bottleneck". Similarly, the overhead in transferring data from PM to CPU is called the "von Neumann bottleneck". Database machines can be categorized according to the techniques used to relieve these two bottlenecks.

The first category of database machines attempts to relieve the I/O bottleneck by moving some of the processing capability from the CPU and placing it closer to the

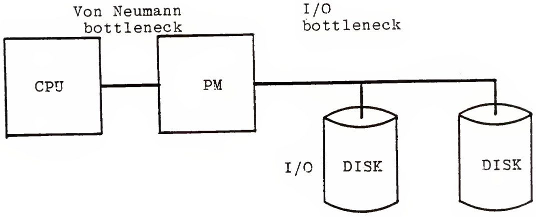


Figure 1.1 "Conventional" Processor Architecture

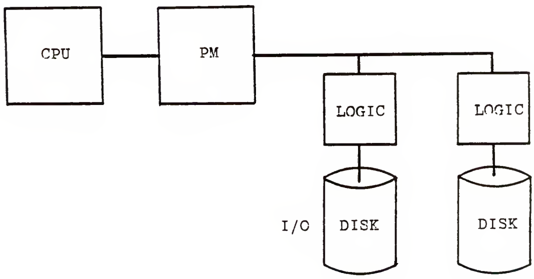


Figure 1.2 Cellular Logic Device

secondary store, as shown in Figure 1.2. These machines are called cellular logic devices; CASSM [SU79], RAP [OZK75, SCH79], RARES [BAT77], etc. are examples of such machines. Cellular logic devices are composed of many "cells" with each cell having the ability to store and process some amount of data. Some of the data is processed in situ and, thus, a lesser amount of data needs to be transferred from I/O to PM for processing by the CPU. The second category of database machines attempts to relieve the von Neumann bottleneck by moving some processing capability into the PM as shown in Figure 1.3. The machines in this category are referred to as associative memory devices [SU80]. The processing capability of associative memories helps in decreasing the amount of data that needs to be staged between PM and CPU. These devices are typically capable of performing some simple, selection type operations.

The third category of database machines are called "backends" [CAN74] and are dedicated DBMS processors connected to the "host" processor bus like any I/O device, as shown in Figure 1.4. Backends are typically full-fledged machines designed to relieve the host computer of all database functions. The host performs only a minimal amount of processing related to the DBMS. Note that the architecture of the backend can itself fit into any one of the categories listed here. The fourth category includes a variety of database processors and special purpose hardware units. The machines in this category will be referred to

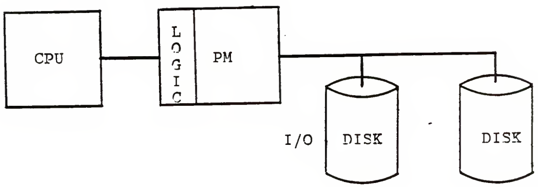


Figure 1.3 Associative Memory System

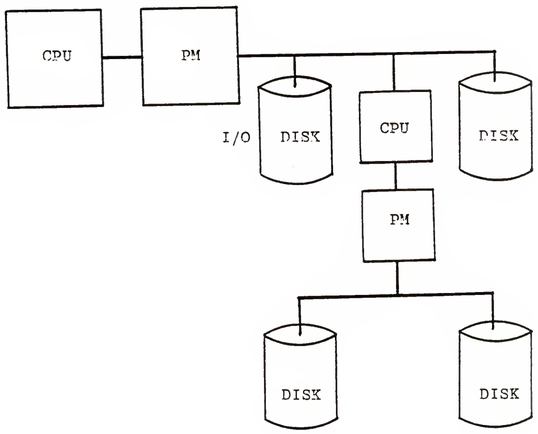


Figure 1.4 Backend Machine

simply as "special purpose processors". Special purpose processors have been designed, for example, for very fast joining [KUN80] and sorting [RAS85] of relations. Database filters [BAN80] fall into this category and are, typically, designed to perform functions like Selection and Projection of tuples. Two types of special purpose processors have been shown in Figure 1.5. In one (box A in the figure), the rate of processing of the processor is compatible with the speed of data transfer on the bus connecting I/O and PM [BAN80]. Thus, no delays are introduced and only a minimal amount of buffering is required. In the other (box B), the special processor is attached to the host processor bus and its processing rate can be independent of the bus speed [KUN80].

More recently, some interest has developed in studying multiprocessor/computer and parallel architectures for database processing as evidenced by [ARD81, MAP81, SHA79, SHA81]. In general, these architectures can support either synchronous or asynchronous SIMD and, in some cases, MIMD processing. Multiprocessor architectures contain multiple processing units connected together via some interconnection scheme. Multiprocessor systems, as shown in Figure 1.6, are "tightly-coupled"; i.e., they are centrally controlled; share system resources like I/O, memory, etc.; and, typically, communicate with each other via shared variables placed in a common (shared) memory.

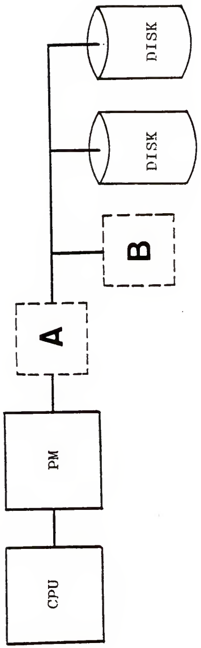


Figure 1.5 Special-Purpose Processors

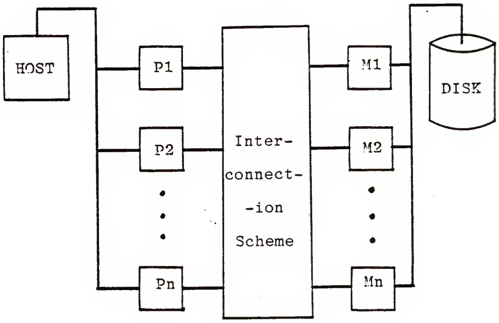


Figure 1.6 Multiprocessor System

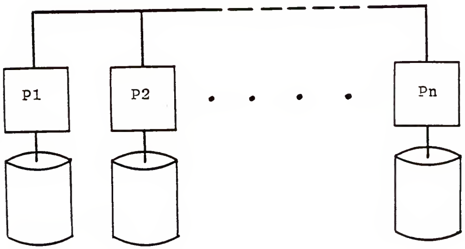


Figure 1.7 MICRONET Schematic

Multicomputer systems, on the other hand, are more loosely-coupled than multiprocessors. They consist of many nodes (computers) which are interconnected by a network. An example, MICRONET [NIC80, SU78, SU83], is shown in Figure 1.7. Each node in MICRONET is a "conventional" computer (specifically, PDP 11/03's were used in the prototype implementation). The interconnection scheme in this case is a common bus local area network which uses a token passing scheme for media access. There is no central control and the system supports asynchronous SIMD processing. Due to the nature of its design, MICRONET cannot support MIMD processing (i.e. the ability to have more than one active global query at a given instant in time).

The easy availability of very powerful and, yet, relatively inexpensive hardware (e.g. 32-bit microprocessors) is a good case for building multicomputer architectures using conventional processors. A system using many conventional processors, if well architected, can perform as well as, if not better than, a system using special purpose processors. The results obtained from simulation of the proposed multicomputer system, and discussed later in Chapters 3 and 4, support this point. This research investigates the performance and control aspects of a multicomputer system in the context of database processing. Such a system should provide architectural features and hardware facilities to support (i) network data exchange, (ii) network communication and synchronization, and (iii) parallel execution of concurrent processes to allow intra- and inter-query concurrencies. The next three sections elaborate on these points.

1.2 Network Data Exchange

In a multicomputer system data files are generally stored in a distributed fashion among many nodes. In performing many common database operations (like the relational join of two global files), large quantities of data have to be moved among these nodes. The conventional approach of moving data is through a data link (e.g., a data bus), typically using input/output instructions. This either ties up the data and control lines and the processors involved in the transfer until the transfer is complete, or burdens the processors involved with interrupt processing, error checking and synchronization tasks. The transfer time can become very significant when large amounts of data are transferred and data movement among nodes often becomes a major bottleneck. We propose to alleviate this problem by using main memory modules which can be switched among nodes. Data to be transferred are stored in these modules and switched to other nodes for exclusive access. Data transfer time is, therefore, reduced to the module switching time. This concept of switchable memory is different, for example, from shared memories used for communication and scheduling [AHU82], global synchronization [MIS82], and creation of virtual address space [SWA77] in that the main memory modules are physically switched between processors and, once switched, are accessed exclusively by different processors without the usual memory contention problem. The memory switching feature introduced here most closely resembles the one used in the MIDAS system [MAP81].

1.3 Network Communication and Synchronization

Since data files in a multicomputer system are typically dispersed across nodes, queries for retreiving or manipulating the distributed files are issued to the entire set or to a selected subset of processors in the system. Two types of operations are necessary to process such queries: (i) communication among the nodes to transmit messages, status and commands, and (ii) synchronization among the parallel processes to ensure orderly and meaningful computations. In conventional packet-switched network systems, communication and synchronization are achieved by forming packets in the source nodes and routing them to the proper destination nodes. This is rather time-consuming since it involves not only packet transfer, packet routing, and interrupt processing, but also the detection of errors in transmission and possible re-transmissions. Multiprocessor systems typically use shared global memories for message transfer and synchronization and suffer in performance due to bus and memory contention even when a small number of processors is involved. The proposed multicomputer system uses hardware means to achieve some of the functions required in network communication and synchronization. A number of control lines are used to provide a means for synchronizing parallel processes.

1.4 Parallel Execution of Concurrent Processes

A database query can typically be represented by a network of serial and parallel branches of sub-tasks. In order to achieve maximum parallelism, one attempts to process these sub-tasks concurrently (intra-query concurrency). Also, query response times may be improved by concurrently executing multiple queries issued by different users (inter-query concurrency). In order to achieve both intra- and inter-query concurrencies and to increase the system throughput, a system should be able to assign different amounts of resources (processors, memories, etc.) to different query sub-tasks (or queries), depending on the complexity of the sub-tasks (or queries). We propose to support parallel execution of concurrent processes by employing dynamic "physical" partitioning of a common-bus network in order to reconfigure it into a number of independent sub-networks.

The concepts of "physical" partitioning and reconfiguration presented in this paper bear some resemblance to several existing works [ARD81, DEW79, KAR78] but with several important differences. Reconfigurability, introduced in [KAR78], allows computers to vary their word sizes (vertical partitioning) to meet the requirements of the task(s) under execution. In our multicomputer system, reconfiguration is done at a higher level, by allowing the grouping and regrouping of processors, and is employed in order to exploit the large grain parallelism of the tasks at hand.

"Logical" partitioning and dynamic allocation of processors operating on multiple queries were features first introduced in DIRECT [DEW79] and also in the newer dataflow architecture [BOR80]. Similar to these two designs, our system can dynamically reconfigure the network and assign processors to support inter- and intra-query concurrencies. Nevertheless, the architecture and techniques used to achieve this are quite different from those used in [BOR80] and [DEW79]. First, the concept of switchable main memory modules is unique to this system. Second, the system employs "physical" rather than "logical" partitioning of processors in order to form sub-networks. The "physical isolation" of processors reduces interference and interruption among sub-networks, thereby increasing throughput, security, and reliability. Third, each processor in the system has its own secondary storage devices. Databases are stored in a distributed fashion and accessed in parallel by the processors. Thus, each node is an independent computer system capable of local as well as global processing, i.e., a multicomputer system.

Some aspects of memory sharing and network partitioning used here are similar to those used in the MP/C system

[ARD81]. For example, the MP/C system uses switches to "physically" partition processors and the memory modules of each processor form a contiguous memory space. Nevertheless, there are a number of significant differences: (i) the individual nodes of the proposed system are more powerful and independent than those of MP/C, which do not have their own secondary storage devices, (ii) each so-called partition in MP/C has only one active processor as opposed to many in the proposed system, and (iii) the MP/C system does not have the concept of local plus global memory--all memory is either local or global. Essentially, the two systems have very different design goals. MP/C is a distributed architecture approached from a multiprocessor standpoint, whereas the proposed multicomputer system is a distributed architecture arrived at from the standpoint of a network of independent computer systems.

This dissertation is organized as follows. Chapter 2 presents the key architectural concepts employed in the Switchable Main Memory Modules (SM3) system. Following that a description of some of the hardware and implementation details is provided. Chapter 3 discusses the software organization of the system and the simulation results obtained for some typical database operations in SM3 using discrete-event simulation techniques. In Chapter 4, the architecture of the multicomputer SM3 system is compared and contrasted with that of local area networks (LAN´s) on the one hand and shared-memory systems on the other. The simulation programs used in Chapter 3 are modified to obtain results for the LAN case. Chapter 5 is devoted to the issue of algorithm representation and mapping. Experience with the design and simulation of the SM3 system has shown that it is important to evolve a methodology for representing algorithms and architectures for database processing. Such a methodology allows systematic algorithm decomposition for performance evaluation purposes and also provides greater insight into the process of mapping algorithms to architectures. Issues related to data distribution and multicomputer systems are discussed. Finally, Chapter 6 provides a conclusion to the dissertation and outlines possible future work stemming from this study.

CHAPTER 2
SYSTEM ARCHITECTURE

This chapter outlines the architectural concepts that are central to the SM3 system. This is followed by a description of some details of construction of the related hardware.

2.1 Architectural Concepts

The three key architectural features of the SM3 system are (i) switchable memory, which is used for supporting data transfer, (ii) a dynamically partitionable bus, which supports MIMD processing in the system, and (iii) global control lines, which are used for inter-processor communication and synchronization. These three features along with their implementation, via a set of status words, are all described in this section.

2.1.1 Switchable Memory

Let two processors P1 and P2, each with its own local memory, share a common main memory module via a dual-throw switch SMS as shown in Figure 2.1. This main memory module can be mapped into the address space of either processor P1 or processor P2. When the SMS is in switch position A, the

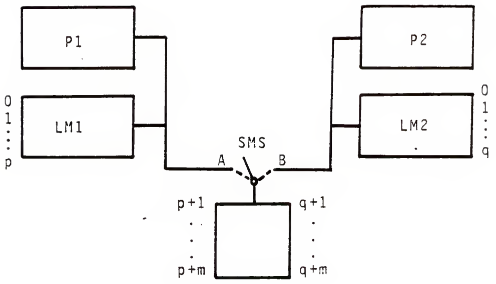


Figure 2.1 The Switchable Memory Concept

SM becomes part of processor P1 which can either read from or write into the module using any memory reference instructions available to it. When the switch is in position B, P2 assumes control over the memory module. Data transfer between the processors can be achieved simply by writing data into this memory module and controlling the position of the switch, thereby reducing data transfer time to the switching time of the switch SMS. Only one of the two processors can exclusively access the memory module at any time. This is fundamentally different from a "shared" memory where both processors would be allowed access to the module by queueing their requests.

The switchable memory (SM) module and the local memory (LM) modules occupy different address spaces. In the figure, the local memories of the two processors are mapped into the address spaces 0..p and 0..q, respectively. Assuming the switchable memory has a size of m, the SM module would then be mapped into address locations p+1...p+m for P1 and q+1...q+m for P2. In the most general case, the SM module may be mapped into a different address space in each processor. However, for the sake of uniformity and ease of implementation, we assume that all SM modules occupy the same address space in every processor. Processors, P1 and P2, have their own local (or private) memories and can function with or without the SM module.

This basic concept of a switchable memory can be extended to build a network of processors which use SM

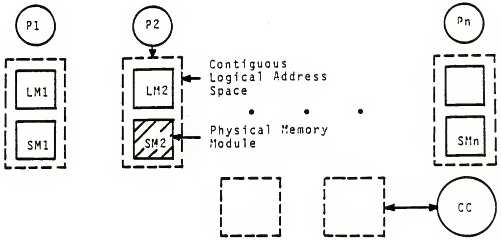


Figure 2.2 Switchable Memories in a Network

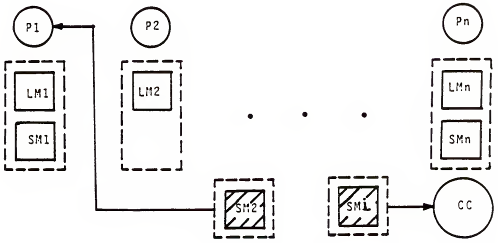


Figure 2.3 SM2 switched to P1 and SMi switched to CC

modules to transfer data among themselves. Figure 2.2 shows such a network of processors where each processor has local as well as switchable memory modules. A global address space is available to all processors. A second global address space is available to a Control Computer, CC, the function of which will be explained later. If a processor, say P2, needs to transfer data to P1 then it would first switch its memory module SM2 into its own address space and write the data into this module (Figure 2.2). It would then switch SM2 into the globally accessible address space (Figure 2.3) so that P1 can access the data in SM2.

Figure 2.3 also indicates that while P1 is accessing data from SM2, the CC can simultaneously access data from some other SM module, say SMi. The SM3 system is centrally controlled by the CC which is responsible for compiling global queries, overseeing network partitioning, collecting outputs, etc. The CC is connected to all the SM modules via a separate Switchable Memory Bus (SMB). Thus it can access the SM modules independently of the other processors in the system.

Broadcast communication in the system is supported by a simple extension of the above switchable scheme. Data can be broadcast from a single processor, say P1, to all the other processors by simultaneously switching all the SM modules into the common global address space, as shown in Figure 2.4. A single write instruction issued by P1 in this address space writes data into all the physical memory

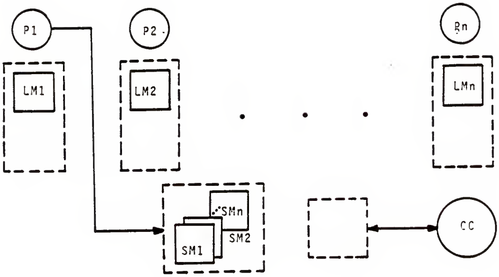


Figure 2.4 Broadcast from P1

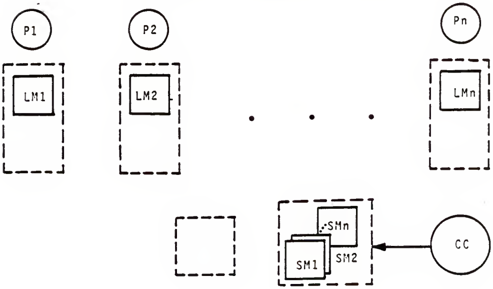
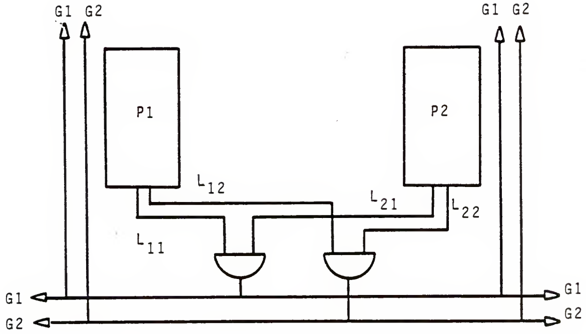


Figure 2.5 Broadcast from CC

modules, thereby achieving broadcast communication. Similarly, the CC can also broadcast data to the processors by switching the SM modules into the address space accessible to it, as shown in Figure 2.5. Our analysis of algorithms in this system has shown that the switchable memory scheme can be best exploited for the purposes of broadcasting data and collecting results. This reiterates the point that the switchable memories are best suited for supporting data transfer operations.

2.1.2 Global Control Lines

Global control lines have been adopted in SM3 from the MICRONET system [NIC80, SU78, SU83]. They allow processors to synchronize their execution; they also provide efficient interprocessor communication. Each processor has five "local" control lines, just as in MICRONET, which are globally AND-ed together to form the five "global" control lines (see Figure 2.6). When a local line is set in all the processors, the corresponding global line will be set automatically. Hence, if a processor in the SM3 system is waiting for the other processors to complete a particular action, it can simply sense a predetermined global line. Processing can then be synchronized and continued, or an interrupt can be issued to the CC. Thus, a high-level protocol for processor synchronization is implemented via the control lines.



$G1 = L_{11} \bullet L_{21}$

$G2 = L_{12} \bullet L_{22}$

Figure 2.6 Schematic of Global Control Lines

The specific usage of the control lines is dependent upon the algorithm being executed. For example, a control line used to indicate the end of the operation in a Select algorithm can also be used to indicate the end of one iteration of the nested-loop Join algorithm. Thus the meaning associated with each line can change according to the software being executed.

2.1.3 A Partitionable Bus

Two processors, say P2 and P4, can make simultaneous requests to transfer data to, say P1 and P3 respectively, as shown in Figure 2.7. Since there is only one global address space for switching in and out SM modules, P2 and P4 obviously contend for this space. This problem of processors contending for the same global address space is solved by dynamically partitioning the bus to which the SM modules are connected. A separate global space is created for each bus partition, as shown in Figure 2.8. Thus, each partition can independently support data transfer operations among its processors.

The dynamic partitioning scheme described above creates independent groups or clusters of adjacent processors in the system. The global control lines are also partitioned along with the processors and can be used independently by each cluster. Thus, the partitioning feature aids in supporting both intra-query concurrency (all the processors in a cluster work on the same database command) and inter-query

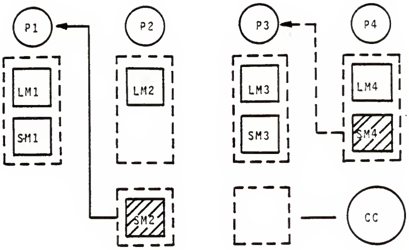


Figure 2.7 Simultaneous request for SM space from P1 and CC

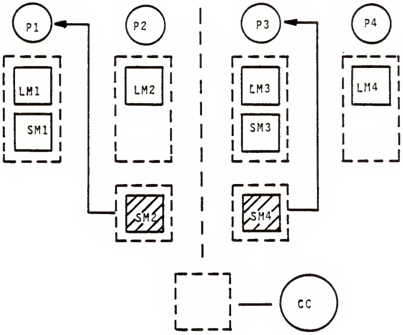


Figure 2.8 Partitioned Network

concurrency (all the processors in a cluster work on the same query). The method of partitioning outlined here requires the "physical adjacency" of data. If data are dispersed over many non-adjacent processors, then either a large cluster should be formed containing some processors that do not have the required data or the data should be moved around in the network to ensure physical adjacency. We believe that physical adjacency is not a serious constraint since (i) data can be initially loaded into adjacent processors and (ii) data can be moved at high speeds using the switchable main memory facility to make them adjacent.

2.1.4 Status Words and Status Word Address Space

A set of status words is used in SM3 in order to implement many of the functions described above, e.g. switching of memories, partitioning of the bus, forming of clusters, etc. A detailed description of the functioning of these status words is provided in Section 2.3.3. Here we shall briefly describe the overall organization of these status words. Each node is associated with five status words. A Node Status Word, NSW is used to control the various switches at each node. Every SM module is associated with a status word which is used to ensure proper access to the module. Since two SM modules are used per node, in order to provide overlap between data transfer and CPU operations, there are two such status words per node,

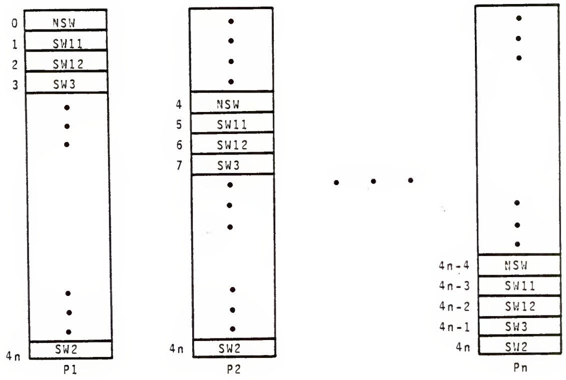


Figure 2.9 Status Words Address Space

SW11 and SW12. The status of NSW, SW11, and SW12 is available in a read-only status word, SW3. Finally, each node is also associated with a status word, SW2 which is used in broadcast communications.

The status words are memory-mapped into a global address space which is accessible to all the processors. Thus any processor may access the status words of any other processor. The status word address space is shown in Figure 2.9. All status words, except SW2, are mapped to distinct addresses in this space. Thus, they can all be individually accessed by any processor (including CC). The SW2's are all mapped to the same address. Thus, setting a bit in SW2 will effectively set this bit in all the processors. This feature is useful in broadcast communication, for example, in order to simultaneously switch the SM modules to and from all the local processors. The status words address space is implemented in shared memory, as explained in Section 2.3, since more than one processor can attempt to access it at the same time.

2.2 Network Communication Modes

The possible communication modes in a general network configuration with N processors controlled by a single control computer are shown in Table 2.1. For example, the control computer can establish one-to-one communication with a single network processor in order to send special messages to it or to test its status, etc.(row 1 of Table 2.1). It

TABLE 2.1
Network Communication Modes

Operation	CC		SM of P1		SM of P2		SM of Pi		SM of Pm		SM of Pn	
	Send	Receive	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
1	X							X				
2	X			X		X		X		X		X
3		X					X					
4							X			X		
5				X		X	X	X		X		X

can also invoke the one-to-all communication mode in order to broadcast data/commands, "bring up" the system and the like (row 2 of Table 2.1). On the other hand, a local processor Pi may have completed an operation (like MIN, MAX, COUNT, etc.) and may want to transfer the result to CC; it may then establish a one-to-one communication with CC (row 3). If Pi needs to communicate with another processor Pm, in order to transfer data or messages, then it would establish one-to-one communication with Pm (row 4). Pi can also establish one-to-all communication (required in operations like JOIN, statistical aggregation, etc.) with all the other processors of the cluster (row 5), in order to facilitate intra-cluster communication. As described later, in this mode the broadcasting processor Pi is called the cluster control processor (CCP) and it inherits most of the properties of the CC.

Since data transfer is via the main memory modules, in order to support the above communications requirements, the architecture should (i) allow the SMi´s to be switched between the Pi´s and CC (ii) allow all SMi´s to be mapped to the same address space to implement the one-to-all communication mode, (iii) allow data to be read from only a single module when all the modules are mapped to the same address space, and (iv) synchronize accesses to the switchable memories and ensure that any contention is resolved. These features are supported by the status words described above in 2.1.4 and discussed in more detail in Section 2.3.

2.3 Architecture Details

The node computers of SM3 are stand-alone systems with their own processing unit Pi, some local main memory LMi, a switchable main memory module SMi, secondary storage devices, terminals and the like. Data are transferred between disk and CPU via dual I/O buffers which are, in fact, memory modules that are switchable between the I/O controller and CPU. Each node also has dual SM modules, rather than one module, in order to provide an overlap between data transfer and CPU operations. The set of status words associated with each node serves to control network switches and access rights to the SM modules. As described above, the status words are implemented in a physically distributed, globally shared address space. A status word, SW1 is associated with each SM module and the status words SW2, SW3, and NSW are associated with each node of the network. The following symbolic notation will be used throughout in order to refer to the various system components:

CC	--	control computer
Pi	--	a local processor
CCP	--	cluster control processor
SM1i, SM2i	--	two switchable memory modules at Pi

LMi	--	local memory of Pi
SMSi	--	switch used for connecting SMi to Pi, CCP or CC
CCPSi	--	cluster control processor switch for Pi; designates Pi as current CCP
Si	--	switch associated with Pi, used for forming clusters
SW11i and SW12i	--	SW1 status words associated with SM1i and SM2i, respectively
SW2i, SW3i, and NSWi	--	SW2, SW3, and NSW status words associated with Pi
CCB	--	cluster control bus
SMB	--	switchable memory bus

A schematic of the SM3 multicomputer system is shown in Fig. 2.10. The system has two distinct buses: a Switchable Memory Bus SMB, which connects all SMi modules in the system to CC and a Cluster Control Bus CCB, consisting of address, data and control lines and also the global control lines described in Section 2.1.2. The SMB is not partitionable and is used for transferring data between Pi and the CC while the CCB is partitionable and is used solely for intra-cluster communication. The use of two buses allows the system to support multiple clusters operating concurrently.

2.3.1 Node Level Architecture

In this sub-section we shall first describe the bus organization in each node of the SM3 system and provide a

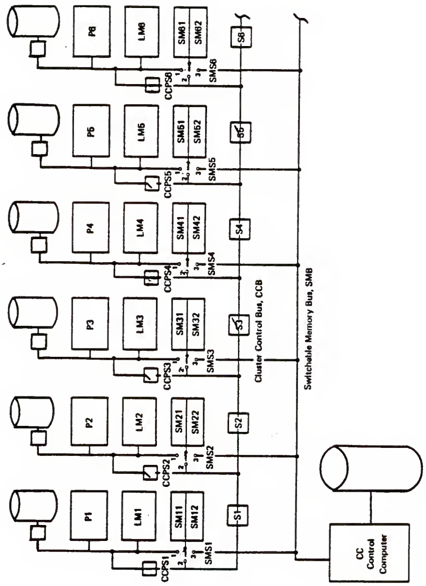


Figure 2.10 The Switchable Main Memory Modules System

more detailed description of the node level architecture. As shown in Figure 2.10, the SMB is connected to the CC and runs system-wide as does the CCB which can be partitioned using the Si´s. The internal bus of each Pi connects the CPU, memory, and other devices at each node. All three buses, SMB, CCB, and the internal bus, have access to the status word address space and to the switchable memories. The switches at each node consist of the Si, the CCPSi, and the SMSi which is used to switch the SM module into one of four positions--local, cluster, global, or maintenance mode (see 2.3.2).

Figure 2.11 shows the above bus organization in greater detail for a single network node. Address references on the internal bus originate from Pi and those on the SMB originate from CC. References on the second bus (CCB) may originate either from CCP or from the local Pi. When a node is designated as a simple processor (Pi) this bus is connected to the CCB and, hence, the references originate from the CCP. When a node is itself the CCP, then the internal bus is connected to the CCB and they carry identical data. This arrangement requires special attention while accessing the status word address space, as described below in 2.3.2. Figure 2.11 also shows the Si and CCPSi switches controlled by lines from the SM module interface unit. The interrupt line from Pi to CC is sent via the SM interface unit to the SMB. Similarly, the interrupt from CC to Pi is sent to Pi via the SM interface.

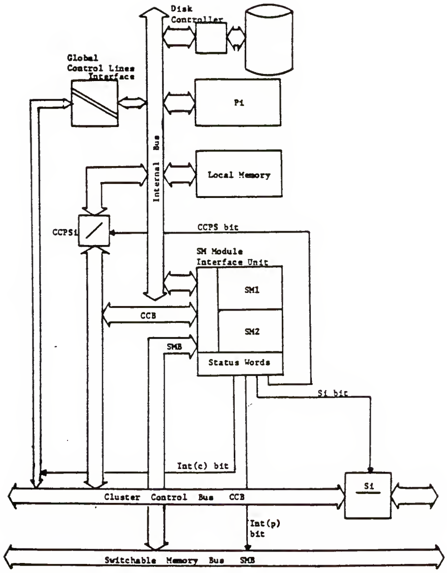


Figure 2.11 Bus Organization at a Node

In addition to the interrupt from CC, the five global control lines of CCB (see 2.1.2) are also connected to Pi via the MICRONET-like bus interface called the global control lines interface. The MICRONET interface is built in two parts [NIC80]. One part interfaces to the control lines and the other to the system bus. Hence, the control lines may be connected to any computer system by changing only the system-specific part of the interface.

2.3.2 SM Module Interface

We shall now explain the SM module interface unit shown in Figure 2.11 in more detail. The detailed block diagram of the interface unit is shown in Figure 2.12. The input to this unit consists of the three system buses each containing address, data, and control lines and the output consists of two buses--one to SM1 and another to SM2--along with the various control lines from the status words. The interface contains the status words along with the logic to provide shared memory access to them; the switching logic for switching two out of the three buses to the two SM modules; the logic to implement broadcast communication; and other relevant hardware.

The interface hardware may be divided into two sections--one to manage accesses to the status words and another to provide access to the SM modules. The higher-order bits of the incoming address lines are decoded to determine if the status word address space is being

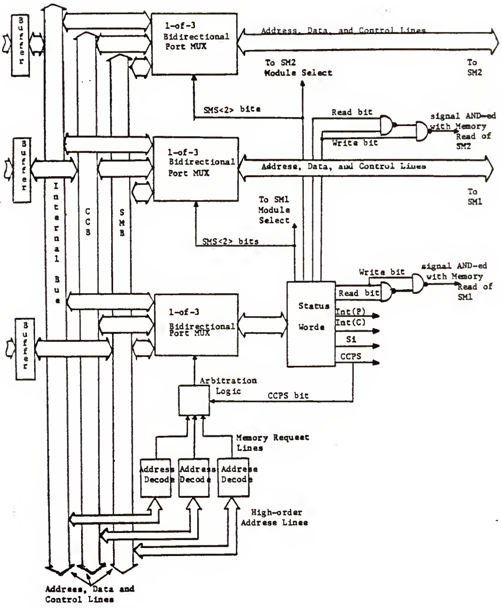


Figure 2.12 SM Module Interface

accessed. Since the status words are mapped to a globally unique address space, any processor may access any status word. Requests from the three input buses to the status word space are queued in sequence by the arbitration logic which also resolves simultaneous requests by using a random assignment logic. The bus with the first memory access request is switched to the status words by a 1-of-3 port multiplexer unit. Since conflicting requests are resolved at the hardware level, the software as noted in Section 2.4, needs to read back a status word after writing into it in order to ensure that the write was successful.

The two SMS bits of SW1 and SW2 determine which one of the three input buses accesses a given SM module. The three buses are again multiplexed via a 1-of-3 port multiplexer. The SMS bits are also used to "tri-state" an SM module by OR-ing them together and connecting the output to the module-select (or chip-select) pin of each SM module. Thus the SM module is deselected when both the bits are zero. When a node is designated as the CCP its internal bus is connected to the CCB. The internal bus and CCB carry the same memory references thus generating duplicate requests to the status word address space. The duplicate request problem is handled by the CCPS bit from NSW (which is set to 1 when a node is the CCP) which adjusts the arbitration logic so that only one of the two requests is accepted and the other ignored. Note that there is no problem with duplicate requests while accessing the SM modules since, in this case, one out of the three input buses is switched explicitly by the SMS bits. This illustrates a fundamental difference between a "shared memory" and a "switchable memory".

The hardware also contains logic to support the Write and Read bits of SW1 and SW2. The read control signal of the SM module is AND-ed with a control signal from the interface unit so that when the Write bit is set, it is neccesary for the Read bit also to be set if data are to be read from a module. The interface unit also supports the CCPS, Si, and interrupt signals in each node. The interrupt from Pi to CC is derived from the INT(p) bit and is daisy-chained along the SMB to CC. The interrupt to Pi is derived from INT(c) and fed via the global control lines interface (see 2.3.1) to the internal bus.

The decode and arbitration logic and the bus multiplexers of the interface hardware, in conjunction with the status word bits, provide true shared memory access to the status word address space and switchable memory access to the SM modules. The appropriate bus, with all the address, data, and control lines, is switched to each SM. In addition, the hardware also supplies the processor interrupts and the control lines to control the various switches. Using this hardware the SM3 system can, for example, be configured as shown in Figure 2.10. All the SMSi switches are in cluster mode (SMS<2>=10); CCPS1, CCPS4, and CCPS6 are closed (CCPS=1); and S3 and S5 are open (Si=1).

Thus, three clusters are formed consisting of 1) P1, P2, and P3, 2) P4 and P5, and 3) P6 to Pn, with P1, P4, and P6 desi~nated as the respective CCP´s.

2.3.3 The Status Words

Each switchable memory module is associated with a status word SW1 which is used to record the current status of the module in the one-to-one communication mode. Since two switchable memory modules are used per node, there are two status words SW11 and SW12. Each node is also associated with a status word SW2 which is used only in the one-to-all communication mode. The current status of both memory modules is available in the status word SW3 which derives its bit values from SW11, SW12, and SW2. The SW11´s, SW12´s, and the SW3´s are mapped to distinct addresses in the status word address space, thereby enabling them to be addressed individually by either the Pi´s or the CC. The SW2´s, on the other hand, are all mapped to the same address and are used to implement the one-to-all communication mode, as explained below in Section 2.3.3.2. Finally, each node has a Node Command/Status Word, NSW which is used to set/reset the Si and CCPSi switches at a node and to indirectly specify which one of the two SM´s is to be used in broadcast communication. All processors have "write-only" access to all the SW11´s, SW12´s, and SW2´s; "read-only" access to all the SW3´s; and "read/write" access to all the NSW´s. As described in Section 2.1.4, the status words are memory-mapped and the status word address space, which is unique to the entire network, may be viewed as shown in Figure 2.9.

The bit assignments and interconnections for the status words NSW, SW11, SW12, SW2, and SW3 are shown in Figure 2.13. The following sub-sections give a description of the function and operation of each of these bits.

2.3.3.1 The Node Status Word, NSW

The NSW maintains information pertaining to the node switches, the interrupts, and the broadcast mode of communication. The Si, CCPS, and the two BRD bits are read/write bits and may therefore be set or reset by a processor. The SMS1, SMS2, INTp, and INTc bits are read-only bits and provide only status information. The Si bit controls the Si switches which are used for partitioning the bi-directional CCB in order to form clusters in the network. The CCPS bit is used at the time of cluster formation to designate one of the processors in the cluster as the cluster control processor (CCP). The CCP assumes most of the control responsibilities of the CC. The internal bus of the CCP is connected to the CCB via the CCPSi switch to allow broadcast communication within a cluster (note that only a CCP can broadcast data within a cluster). The CCPSi switch connects/disconnects the internal bus to/from the CCB and is controlled by the CCPS bit.

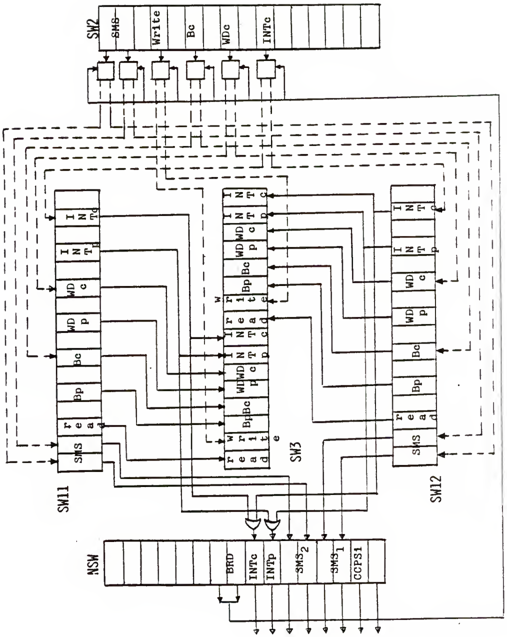


Figure 2.13 Status Words Interconnections

The two BRD bits are used to specify the SM module to be used for broadcasting data. In a cluster of, say N processors, it is not always possible to guarantee that every node will use the <u>same</u> SM, say SM1, in order to transfer data in the broadcast mode. Thus, the BRD bits are introduced to allow each node to independently specify which of the two SM's it intends to use for this purpose. As shown in fig. 2.13, status word SW2 contains only those bits that are relevant for broadcast communication. The bits in SW2 are routed to either SW11 or SW12 under control of the BRD bits from NSW. The BRD bits are normally set to 00 in order to tristate the outputs of SW2. In other words, when BRD = 00 the SW2 is isolated from both SW11 and SW12, and both SM's can be used for data transfer in the one-to-one communication mode under control of SW11 and SW12, respectively. When the BRD bits are set to 01, the bits from SW2 are routed to SW11 and any changes to SW2 (which can occur only during broadcast communication, as mentioned in 2.3.3) affect the corresponding bits in SW11. Thus, BRD = 01 implies that SM1 is to be used in broadcasting data. Similarly, when BRD = 10 the bits from SW2 are routed to SW12, and the implication is that SM2 is to be used in broadcasting data. The BRD bits are set before each broadcast and reset at the end of broadcast.

The read-only bits SMS1, SMS2, INTp, and INTc provide status information related to the two memory modules (SM1 and SM2) and the node interrupts. The SMS bits indicate the current connection of each SM module (see 2.3.3.2 below). The INTp (INTc) bit indicates that an interrupt has been issued by Pi(CC) to CC(Pi) and invokes interrupt service routine. Since the INTp and INTc bits in SW11 and SW12 are OR-ed and fed to the NSW, a processor needs to read SW3 in order to determine which SM module the interrupt pertains to. The interrupts from the Pi´s to the CC are daisy-chained and the CC has to step through each node and read the NSW to determine which particular node sent the interrupt. An interrupt is issued only by the sending processor to inform the receiving processors of incoming data.

2.3.3.2. Status Words SW1 and SW2

Status word SW1 is used for one-to-one communication, whereas SW2 is used for broadcast communication. Thus, SW2 contains only those bits which are relevant for the broadcast mode of communication. Since all the SW2´s are mapped to the same address, the bits in all the SW2 words (either in a cluster or in the entire system) can be set/reset by a single write instruction. We shall now describe the various bits in the status words SW1 and SW2.

The SMS bits of SW1 and SW2 control the operation of the memory module switches. The two bit SMS field allows four settings. In each of the settings the memory is connected as follows:

00 -- memory is tri-stated

01 -- connect to internal bus (local)

10 -- connect to CCB (cluster)

11 -- connect to SMB (global)

Setting these bits in SW2 affects all the modules in the cluster (or system), whereas setting the bits in SW1 affects only the memory module of the given node. As stated above, the SM module affected by the setting of bits in SW2 is determined by the BRD bits in NSW.

The Read bit in SW1´s is used in conjunction with the Write bit in order to provide read access to a single, "source" SM module in the broadcast mode of communication. In order to broadcast data, all the SM modules are mapped to the <u>same</u> address space and the Write bit is set. Thus, a memory write issued by the CCP/CC has the desired effect of writing into all the SM´s. At the same time, a memory read would have unpredictable results. The SM interface unit implements logic so that whenever the Write bit is set any read instruction is recognized only by the single SM module in which the Read bit is also set.

The Busy(p) (Bp) and Busy(c) (Bc) bits indicate that the memory module being referred to is currently under active use by Pi or CC, respectively. These bits are always set <u>before</u> the start of a read or write operation and reset <u>after</u> the completion of the corresponding operation. The Write_Done(p) (Write_Done(c)) bit, abbreviated as WDp(WDc), indicates that Pi(CC) has completed writing valid data into the memory module. Hence, the current contents of the module cannot be overwritten by Pi(CC) until WDp(WDc) is reset. This bit is set by the sending processor <u>after</u> the completion of a write operation and reset by the receiving processor <u>after</u> it finishes reading the contents.

The interrupt bits INTp and INTc are set by processors which have data to send out to other processors. The INTp bit is set by Pi and the INTc bit by CC or CCP. The SMS, Bc, WDc, and INTc bits in SW11 and SW12 can be set directly by a processor or indirectly via SW2. In the one-to-one communication modes, the BRD bits in NSW are set to 0 and all the bits in the SW1's need to be set directly. In one-to-all communication the BRD bits are either 01 (SM1) or 10 (SM2). In this case, the bits in the SW1 associated with the SM being used for broadcast, are derived from SW2. This is achieved by the set of multiplexers located at the output of SW2 (fig. 3) under control of the BRD bits from NSW.

2.3.3.3 Status Word SW3

The status word SW3 derives all its bit values from SW11, SW12, and SW2, as shown in Figure 2.13. Each SW3 contains the Read, Write, Bp, Bc, WDp, WDc, INTp, and INTc bits for the two memory modules. All the bits (except the Write bit) are derived from SW11 and SW12 which, in turn, may derive their bits directly via a write instruction from a processor or indirectly via SW2, as stated in 2.3.3.1.

2.4 Usage of Synchronization Bits

We shall now demonstrate how the various status words and their bits are utilized in establishing a uniform switchable memory access protocol. Access to the switchable memories is always preceded by the setting of appropriate bits in the various status words to ensure that there is no memory access conflict. If an SM module required by a processor is already in use, then the processor has to wait for the SM to be free.

The following sub-sections describe two code segments which illustrate the protocol followed by (i) a sending processor that needs write access to the SM, and (ii) a receiving processor that is interrupted by the sending processor.

2.4.1 Sending Processor Protocol

The sending processor first checks whether the SM is free. If so, it sets the appropriate Busy bit and the SMS bits in SW1 and transfers data to the SM until either the data is exhausted or the SM is full. The SM is then "tri-stated" and the receiving processor is issued an interrupt. We shall use the notation SW1ki to mean the SW1 status word of the kth memory module (k = 1, 2 for SM1, SM2) of the ith network node. For SW2, SW3, and NSW we shall use SW2i, SW3i, and NSWi to indicate the corresponding status words of the ith node. Also, the various bits of SW3 (Busy, Write-

```
/*********************************************************/
/*                                                       */
/*  Software in  Pi to obtain  write access to SMk.  For */
/*  the software in CC/CCP to obtain write access to SMk */
/*         change all (c) to (p) and  vice-versa.        */
/*                                                       */
/*  Since the status words are in  multi-ported memory a */
/*  WRITE into the status word address space is followed */
/*    by a READ to ensure that the WRITE operation was   */
/*                  performed  succesfully.              */
/*                                                       */
/*********************************************************/

repeat              /* -- until all the data is transfered   */
  repeat            /* -- until Pi has gained access to SMk  */
    repeat          /* -- until SMk is free for access       */
      READ SW3i(Busy(c)k, WD(p)k, WD(c)k)
    until (Busy(c)k = 0) AND (WD(p)k = 0) AND (WD(c)k = 0);

    WRITE SW1ki(Busy(p) := 1, SMS := local);
    READ  SW3i(Busy(p)k, SMSk)
  until (Busy(p)k = 1) AND (SMSk = local);

  repeat
    Write data into SMk;
  until (End of Message) OR (SMk is full);

  WRITE SW1ki(Busy(p) := 0, WD(p) := 1);
  WRITE SW1ki(INT(p) := 1, SMS := tri-stated);
until (End of Message);
```

Figure 2.14 Sending Processor Protocol

```
/**********************************************************/
/*  Interrupt routine in CC/CCP to read data either from  */
/*                   SM1 or from SM2.                     */
/*  For routine  in Pi -- change all  (p)'s to (c)'s and  */
/*                      vice versa.                       */
/**********************************************************/
READ NSW(INT(p));
if (INT(p)<>1)
   then (Try next node, if CC or CCP.)
else
   begin
      READ SW3i(Busy(p)1, WD(p)1, INT(p)1);
      READ SW3i(Busy(p)2, WD(p)2, INT(p)2);
/**********************************************************/
/*  Check if data is in SM1 and, if so, verify status of  */
/*  bits in SW1. If status is OK, set status to indicate  */
/*  that SM1 is in use.  If not, then error in accessing. */
/**********************************************************/
      if (INT(p)1=1)
         then if (Busy(p)1=0) AND (WD(p)1=1)
                 then begin
                        WRITE SW11i(INT(p):=0, Busy(c):=1,
                                    SMS:=global/cluster);
                        repeat
                          Read data from SM1
                        until (End of Data);
                        WRITE SW11i(Busy(c):=0, WD(p):=0)
                                          SMS:=tri-stated)
                      end
              else ("Error in memory access protocol!")
/**********************************************************/
/*  If  interrupt  not from  SM1 then check if from SM2.  */
/*  Check if data is in SM2 and, if so, verify status of  */
/*  bits in SW1. If status is OK, set status to indicate  */
/*  that SM2 is in use.  If not, then error in accessing. */
/**********************************************************/
      else if (INT(p)2=1)
              then if (Busy(p)2=0) AND (WD(p)2=1)
                      then
                         begin
                           WRITE SW12i(INT(p):=0,Busy(c):=1,
                                 SMS:=global/cluster);
                           repeat
                              Read data from SM2
                           until (End of Data);
                           WRITE SW12i(Busy(c):=0, WD(p):=0,
                                            SMS:=tri-stated)
                         end
                   else ("Error in memory access protocol!")
      else ("Error in Interrupts!!");
   end  /* end of ELSE part of  IF (INT(p)<>1) */
```

Figure 2.15 Receiving Processor Protocol

Done, etc.) are followed by a subscript, k (k = 1 or 2) to indicate the two sets of bits for SM1 and SM2, respectively. The psuedo-code which implements the sending processor protocol is shown in Figure 2.14.

2.4.2 Receiving Processor Protocol

The receiving processor is informed about data in the SM via an interrupt. When a processor is interrupted, it first reads the NSW to ensure that the INTp (or the INTc) bit is set. It then reads SW3 to find out which particular SM module contains the data. Thus, the NSW reflects the node interrupt status while the SW3 provides the interrupt status of the SM module. The code segment in Figure 2.15 is representative of the interrupt service routine, in either CCP or CC, which reads data from the SM module.

CHAPTER 3
PARALLEL ALGORITHMS AND SIMULATION

This chapter outlines the implementation of parallel versions of algorithms for certain commonly used database operations. Following this a brief description of the architecture model used in simulation is provided along with an explanation of the parameter values used and the assumptions made regarding data distribution, etc. The results obtained from simulation for the different operations are then discussed. To start with, the assumptions made regarding the software facilities available at each node of the SM3 system are stated.

3.1 General Software Organization

Each processor in the SM3 multicomputer system is a multi-user, stand-alone computer system with its own secondary storage, the required operating system, related utilities, etc. The software required for performing non-local tasks would be repeated in each processor. Queries entered at any one of the local processors in the system may be either local or global. A local query can be fully supported by a local processor; but a global query requires communication with some other processor(s). All global queries are sent to the CC which decides on the processors that should participate in the query execution and correspondingly forms a cluster. The CC optimizes on cluster size by moving isolated segments of data, if any, into processors which are closer to the main body of the data. This reduces the average size of a cluster and allows a higher degree of parallelism in the system.

The simulation programs used here measure the performance of database operations taken in isolation. The operations considered here are all global in nature. The simulation is not concerned with the fact that each node can be a multiprogrammable system and can, therefore, support many local queries even while executing a global query. Also, all attempts at load balancing and minimization of idle processors is from a global or system-level perspective. The loads imposed on processors due to local processing are not accounted for.

3.2 Parallel Algorithm Implementation

3.2.1 SELECT

The SELECT operation starts with the CC forming a cluster of relevant processors and sending the command to the cluster. The processors use a local line (see 2.1.2), say L(1), to indicate the end of local processing. The command is globally complete when the global line G(1) is set. The operation starts with each processor reading a block of the source relation and applying the selection criteria. Dual, switchable I/O buffers allow each block of the source relation R to be read in asynchronously by an independent I/O controller. This permits overlapping of I/O and CPU operations.

The line L(1) is set when the source reaches "end of file (EOF)". The results from the SELECT operation may be required either as temporary data at the local node or as the final output to be routed to CC. In the latter case, the selected tuples are transferred to CC via the dual SM modules. When one module is full (or when the source relation is at EOF), the Pi uses the sending processor protocol shown in Figure 2.14 to switch the SM over to CC and continues with the selection operation until L(1) is set.

3.2.2 JOIN

This is an operation used frequently in relational databases to cross-relate two relations. The JOIN of two relation A and B, over attributes a from A and b from B, is obtained by concatenating each tuple from A with a tuple from B, such that a O b, where O is a member of the set {=, <=, <, >, >=, ~=}. The well-known nested-loop algorithm for JOIN is used to highlight the features of the SM3 system. Relation B is assumed to be the smaller of the two relations and, hence, is broadcast block-by-block to the entire cluster. Each processor JOINs every block of B with all its blocks of relation A.

The control lines are used extensively to synchronize the various phases of the operation. To start with, the CCP reads a block of B and the other Pi´s read a block of A. A phase consists of a broadcast of a block of B by CCP, followed by the local JOINs in each processor. Each phase is synchronized by the use of a control line, say L(1). When all the blocks of B have been broadcast from the current CCP, the next processor is designated as CCP. The time required to JOIN two blocks, in this case, is greater than that required to read a block of data from I/O, thus the I/O controller is always "ahead" of the CPU. Two control lines, L(2) and L(3), are used to separately indicate the end of I/O operations and the end of CPU operations in CCP.

The functioning of each control line is as follows. Initially, all L(1)´s and L(3)´s are set to 0 (G(1) = G(3) = 0) and all L(2)´s are set to 1 except the L(2) line in CCP. After each processor completes joining the current block of B with all its blocks of A, it sets line L(1) = 1. Thus, G(1) = 1 implies that all processors have completed the previous phase. At this point the processor which has its L(2) = 0, i.e the CCP, may broadcast its next block of B. When the I/O controller in CCP reaches the end of B it sets L(2) = 1 (and, therefore, G(2) = 1). This prompts the processor that is to be the next CCP to set its

L(2) to 0 and to start reading its block of B and prepare for broadcast. Finally, the L(3) line is set to 1 in each CCP when it has finished joining the last block of B. Thus, L(3) = 1 implies that the CCP has broadcast and joined all its blocks of B. Therefore, G(3) = 1 indicates the global completion of the JOIN operation.

3.2.3 Statistical Aggregation by Categorization

The statistical aggregation by categorization is an operation in which the source relation is divided into categories on the basis of certain category attribute values and the aggregation operation is applied to the tuples in each category in order to obtain summary valúes for the category [BAR84]. This operation may be performed using three different algorithms, AGG_LIN, AGG_LOG, and AGG_SEQ.

3.2.3.1 The AGG LIN Algorithm

In the first algorithm AGG_LIN the first phase consists of each cluster processor performing the aggregation operation on its segment of the source relation. Thus, all the processors proceed in parallel in this phase. In the worst case, each processor can encounter a source tuple from every one of the, say C, categories that are formed. It is assumed that all the categories are maintained in a sorted fashion using a memory resident data structure and that the aggregation is carried out on the data in memory. Thus, the processors in this case should be capable of storing a memory-resident table of C categories. The processors may use a line, say L(1), to synchronize at the end of the first phase and begin the second phase of the algorithm. In the second phase, the CCP reads the individual aggregate values computed by the other processors from their respective SM´s and computes the global aggregate value.

3.2.3.2 The AGG LOG Algorithm

The second algorithm, AGG_LOG involves a minor modification to the first. In the above algorithm, the global aggregation in the second phase is carried out by a single processor, the CCP. In AGG_LOG this phase is modified so that the global aggregation is done by partitioning and clustering the processors such that they follow a binary-tree structured pattern. This obviously increases the parallelism in the second phase and the time taken in this phase is now logarithmic rather than linear in the number of processors. We shall briefly explain how a cluster may be partitioned and re-combined to fit the binary-tree structure pattern.

Figure 3.1 shows eight nodes, P1 through P8, containing data on which a certain operation like Average needs to be performed. The sequence of operations is as shown in the figure. In the first stage, the processors are divided into

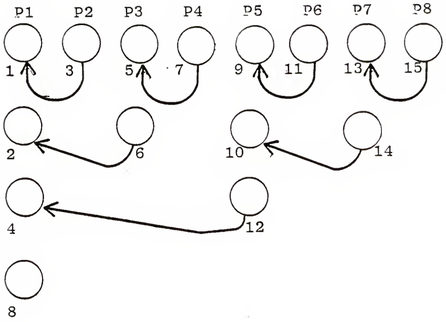


Figure 3.1 Binary Tree Structured Operation Average--Output at P1

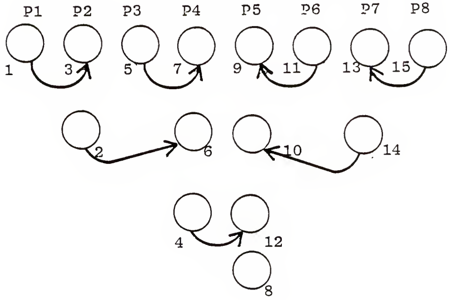


Figure 3.2 Binary Tree Structured Operation Average--Output at P5

four groups of two processors each. One processor in each group accesses data from the other and performs the operation. This stage gives rise to results in four processors viz. P1, P3, P5, and P7. In the second stage, the processors are regrouped to form two groups of four processors each with a single active processor which accesses data from one other processor and performs the operation. Finally, a single group of eight processors is formed and a single processor (in this case P1) performs the final operation. Thus, the final result of the operation is available in P1. The result could be sent to any one of the eight processors by appropriately manipulating the network switches. In figure 3.2 for example, the final result is routed to P5.

The above operations are supported in SM3 as follows. First, it is assumed that all the data required for the operation is available in the SM of each processor. The CCPSi and Si switches are set/reset via the Node Status Word (NSW) associated with each node. The SMS switches are set/reset via the SW1 which is associated with each SM module. Initially, all Si switches are closed and the SMS switches are in position 1. In the first stage, all the inactive processors (viz., P2, P4, P6, and P8) open their Si and CCPSi switches by writing the appropriate control word into NSW. Next, they all switch the SM modules to position 2 by writing a control word into SW1. At the same time, the active processors (P1, P3, P5, and P7) close the CCPSi and Si switches via their NSW´s and set the SMS switch to position 1 via the SW1. The inactive processors set their local lines, say L(1), and the active processors reset their lines at the beginning of each stage. The active processors proceed in parallel and set their L(1)´s when they complete their processing. The end of the stage is indicated when all L(1)´s are set and, consequently, G1 is automatically set to 1. At the end of the first stage, P2 and P6 close their respective Si´s and P3 and P7 open their CCPSi´s (see Fig. 3.1). Again, all inactive processors set their L(1)´s while the active processors (P1 and P5) reset their L(1)´s. At the end of the second stage P4 closes its Si and P5 opens its CCPSi. Finally, P1 is the only active processor in the last stage. The setting/resetting of switches can be done in a regular and pre-determined fashion. From Fig. 3.2 it is clear that data can be routed to any processor by changing the assignment of active/inactive processors.

3.2.3.3 The AGG SEQ Algorithm

The third algorithm, AGG_SEQ is quite different from the above two. Here each of the N processors in a cluster are pre-assigned C/N categories for which they have to perform the aggregation (note: it is assumed that the number and names of the C categories are known a priori). In the first phase each of the N processors reads its source relation block by block and broadcasts it to the entire cluster. All processors scan these blocks of data and pick up those tuples that belong to them in order to form the aggregates. Again the end of the first phase is indicated by the setting of a control line. At this point, each processor has a disjoint subset of the set of all categories. Thus the final results are obtained by merely collecting the aggregate values from each processor.

3.3 Simulation Model

The model of each node used for simulation is shown in Figure 3.3. Each node has a secondary storage device and data is transferred from this device to the CPU via dual, switchable I/O buffers which are also implemented as switchable memory. Dual SM modules are used for transferring data between Pi and CC or among the various Pi´s. Access to the SM modules is controlled via a synchronization variable shown as SW3 in the figure. The model for a single node is repeated as many times as there are processors in the simulation.

The parameter values used in simulation are listed in Table 3.1. The values chosen attempt to reflect a typical environment of operation for the SM3 system and were originally meant to be as close as possible to the values used in [DEW81] so that a reasonable comparison could be made. The actual values would, of course, change depending on implementation. The secondary storage is assumed to be a standard, moving-head disk device. Parameter values for the

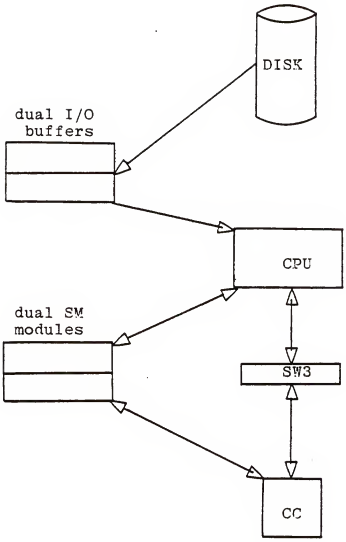


Figure 3.3 Simulation Model of a Node

TABLE 3.1
Parameters Used in Simulation Programs

	DISK PARAMETERS	
BLOCK_SIZE	Size of a single track	13,030 bytes
T	Number of tracks/cylinder	19
TACCESS	Direct-access time	38.6 msec
TREAD	Track (block) read time	16.7 msec
TSEEK	Track seek time	10.1 msec
	CPU PARAMETERS	
TSCAN	Time to perform simple operation on a block of data	12 msec
TMOVE	Time to move a block of data within memory	7 msec
TPROC	Time to perform complex operation on a block of data	95 msec
	SM3 SYSTEM PARAMETERS	
TCODE	Code generation time	200 msec
TCLUS	Cluster formation time	50 msec
TMSG	Interrupt service time	5 msec
TBRDCST	Time to do one-to-all broadcast via shared memories	1 msec
TSWBUFF	I/O buffer switching time	0.1 msec
TSWITCH	Time to set/reset SM System switches	0.1 msec
N	Number of processors per cluster	19
M	Size of buffers and shared memories in number of blocks	1
TUPLE_SIZE	Size of individual tuples in source/output relation	100 bytes
RECS	Number of tuples in source relation	50,000

IBM 3330 disk drive [GOR80] were used. This device has 404 cylinders with 19 recording surfaces per cylinder. Each track on a surface holds 13,030 bytes. The size of a single track will be assumed as the unit of data transfer between secondary storage and primary memory and, also, between switchable memories. The time for a single rotation of the disk is 16.7 msecs. The average direct-access time is 38.6 msec and the track-to-track seek time is 10.1 msecs.

The processing unit is assumed to be similar to a VAX 11/780 CPU with an execution speed of 1 MIPS. Operation times are specified at the block level. The size of a block is equal to that of a track on the disk viz., 13,030 bytes. Assuming that the length of tuples in the relation is 100 bytes, approximately 130 tuples can be accomodated per block. It is assumed that scanning a block of data in order to perform simple operations like selection requires about 12 msec (assuming a 1 MIPS processor). This allows about 100 instructions to be performed per tuple. More complex operations like those required for sorting, joining, and aggregation are assumed to take 95 msecs per block. The time required for moving a block of data within primary memory is 7 msec.

Some basic operations which need to be accounted for in the SM3 system are query compilation and code generation, processor clustering, switching of memories and buffers, and message transfer times. Query compilation and code generation involve steps of parsing, optimization, and the generation of code. A detailed study of such operations was done in [CHA81]. On the basis of the results calculated in [CHA81, DEW79], a time of 200 msecs is assigned for code generation. This figure would be higher if more extensive query optimization is carried out. Once the code has been generated, there is an overhead on the control computer CC to cluster the required processors. This step involves at least two table lookups, one to determine which processors are involved in the operation and the other to check if they are all free. After this the required processors are clustered together by setting the appropriate switches. The time taken in this phase is assumed to be 50 msecs.

It is assumed that all the switches in the network--Si´s, SMSi´s, CCPSi´s--can be set/reset by reading from and writing into a status register. The time required for this is 0.1 msec (equivalent to executing 100 instructions). The time required to switch an I/O buffer between the disk and CPU is also 0.1 msec. The time taken to send interrupt messages between processors via the CCB is 5 msec. Timing values are derived on the basis of the number of instructions required to perform the specific operation (or an equivalent task). Thus, the absolute values could change depending on the speed of the underlying hardware.

Unless otherwise stated, the simulation assumes that all data and attribute values are uniformly distributed across processors. Based on this assumption, whenever the size of a cluster is increased for a given relation, the number of blocks per processor of the relation decreases. Similarly, the specified selectivity factor is applied equally to all the processors in a cluster.

3.4 Results of Simulation

An analytical evaluation of the SM3 system using timing equations was carried out in [BAR83]. The objectives behind this analysis were (i) to provide a comparison with other systems/architectures and (ii) to identify activities which consume more time than others, with a view towards optimizing overall response time. The details of the performance evaluation using timing equations and comparison with other systems are available in [BAR83, SU84]. Results obtained from the simulation experiment are now presented.

Simulation programs were written in the C language under UNIX for the SELECT, JOIN, and Statistical Aggregation operations using the discrete-event simulation technique. A simulation study, in general, provides a better approximation of the system than timing equations. The simulation model used for each node of the SM3 system has been described above. The algorithms for SELECT, JOIN, and Statistical Aggregation are implemented as described in sections 3.2.1, 3.2.2, and 3.2.3, respectively.

The simulation results for the SELECT operation show the same trends as shown by a previous timing equation analysis [SU84]. Figure 3.4 shows a comparison of results obtained from simulation versus those from timing equations.

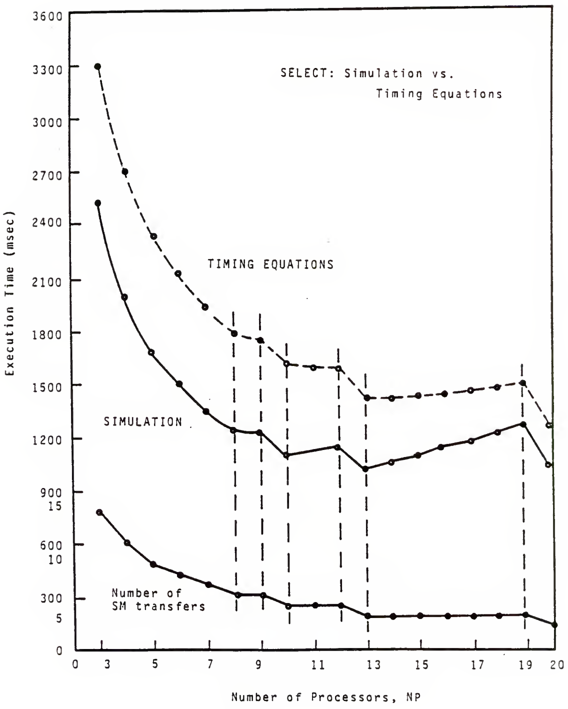
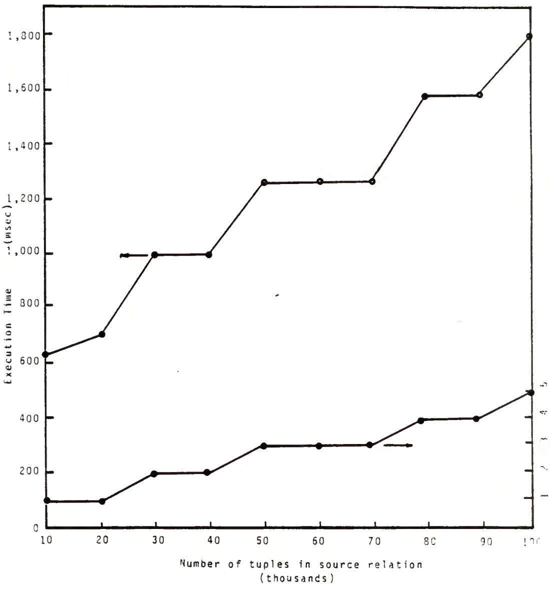


Figure 3.4 SELECT: simulation vs. timing equations. Selectivity factor = 0.1, Relation size = 50,000 tuples.

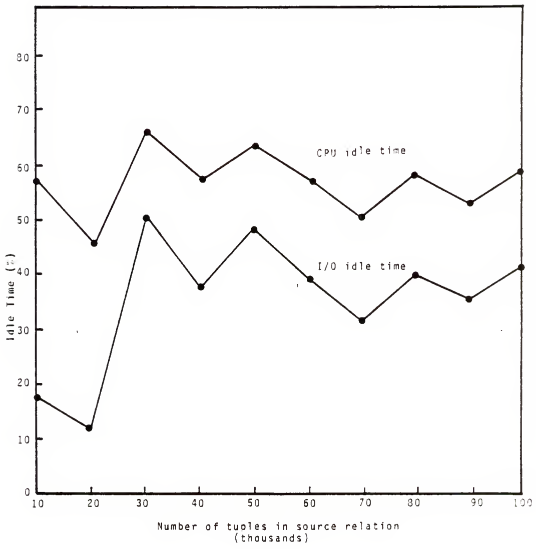
The X-parameter, the number of processors in a cluster, is varied from 3 to 20 for a given source relation size and selectivity. A key factor that determines operation time is the number of SM transfers between Pi and CC. The number of SM transfers is the ceiling function of the result obtained by dividing the size of the output relation at Pi (in blocks) by the size of the SM module (in blocks). Thus, if the SM module size is 1 block then, whether the output is 1.09 or 1.99 blocks, the number of transfers is still 2.0. Thus, the figure shows the number of SM transfers remaining constant for cluster sizes in the intervals (8, 9), (10, 12), and (13, 19). For these intervals the operation time increases with increase in cluster size. For the intervals (3, 8), (9, 10), (12,13), and (19, 20), the operation time decreases since there is a decrease in the number of SM transfers. Thus, for a fixed source relation size, increasing the number of processors in a cluster (assuming uniform data distribution) would increase operation time unless there is a decrease in the number of SM transfers.

The fact that the SELECT operation is more sensitive to the result relation size than to the source relation size is illustrated in Figure 3.5(a), where the operation time is plotted against source relation size (fixed selectivity). For source relation sizes (in tuples) in the intervals (30,00, 40,000), (50,000, 70,000), and (80,000, 90,000) the operation times remain the same. In these cases, as shown in the figure, the number of SM transfers required per node to



(a)

Figure 3.5 Effect of source relation size on execution time. a) total time and number of SM transfers; b) CPU & I/O idle time percentages.



(b)

Figure 3.5--continued

transfer the results to CC is the same even though the source size has increased. Thus, the output size has a greater impact on operation time than the input size. Also, as shown in Figure 3.5(b), in these cases the CPU and I/O idle times decrease since, even though the operation time remains constant, both I/O and CPU have to do more work in order to read and process the larger source relations.

The output from the SELECT operation is routed back to CC via the SM modules. The CC services SM requests from the Pi´s using a round-robin (RR) schedule. Thus, if CC is currently servicing the SM of, say node P4, and if SM´s from P1, P3, and P5 are queued for service, they will receive service from CC in the order--P5, P1, and P3. A possible modification is to adapt the CC service schedule to the load offered by each node. Rather than employing a simple RR scheme, the CC can first service nodes in which <u>both</u> the modules are filled with data (i.e., nodes blocked due to the unavailability of SM´s) and then service all the pending nodes.

The modified RR scheme has a slight effect on operation time only in cases of highly unbalanced output, i.e. where a few nodes of the cluster account for a large part of the output. In the simulation, the selectivity factor of a single node (of a 19-node cluster) was varied from 0.1 to 1.0. Figure 3.6 shows the effect of using a plain RR scheme versus an RR scheme modified as stated above. For lower selectivities, i.e. 0.1 and 0.2, the service pattern has no

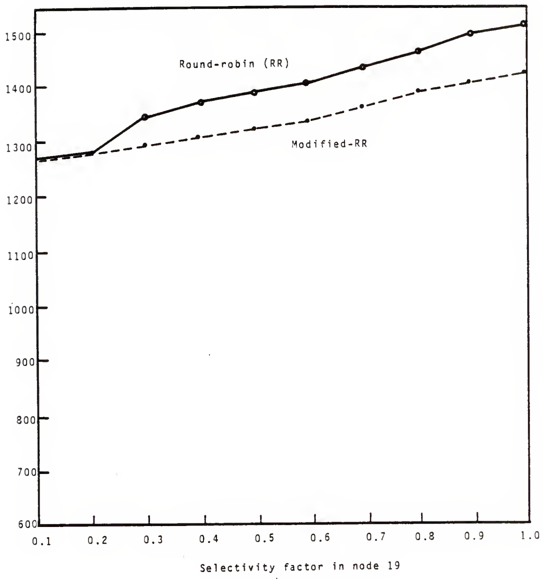


Figure 3.6 Effect of service pattern on execution time. Number of processors = 19; source relation size = 50,000 tuples.

effect on operation time. For higher values of selectivity, though the modified RR scheme performs better than the plain RR scheme, the improvement is only in the range of 5% to 10%.

Since JOIN is an inherently more complex operation, the timing equations approach is not adequate enough to capture all the details of the operation. This is clear from our simulation results which are not very close to the results from the timing equation analysis, unlike the SELECT case. Figure 3.7 shows the plots for the JOIN operation. The discrepancy in the results are attributed to the following factors. First, the simulation was able to represent, much more realistically, the sharing of the dual I/O buffers between blocks of relations A and B, unlike the timing equations which made simplifying assumptions in this regard. Second, the assumption that I/O operations are totally overlapped by CPU computations is not always true. When a block of B is broadcast, the CCP needs to wait on I/O for the next block of A (since broadcast time is less than I/O access time), thereby adding to the overall time. The operation can be easily optimized at this point by "re-using" the last block of A for joining with the current block of B, as suggested in [KIM80].

A point to be noted is the definition of selectivity factor in the case of the JOIN operation. In [DEW81] and [SU84] the maximum output of the JOIN operations was estimated to be twice the product of the number of blocks in

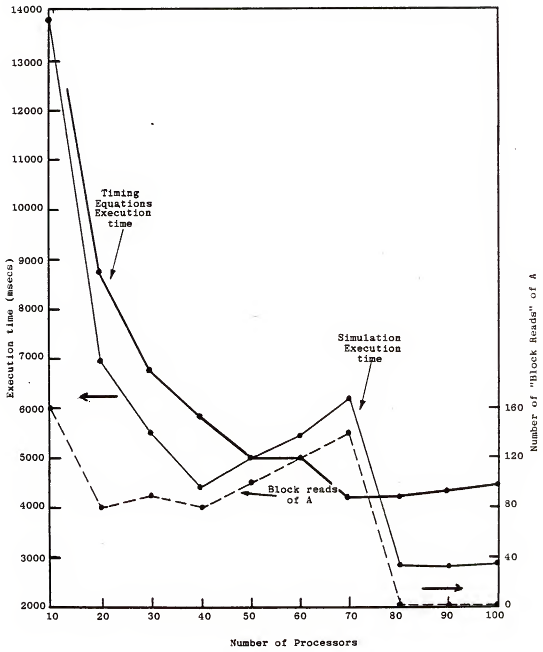


Figure 3.7 JOIN execution time vs. number of processors. Relation A = 10,000 tuples, relation B = 3,000 tuples.

relations A and B. The factor of two accounted for the concatenation of tuples. The selectivity factor was defined as the size of the output expressed as a fraction of this maximum size. The method used for calculating the maximum size of the output actually results in an underestimation. The worst case output of a JOIN operation is the cross-product of the two relations. Thus, the maximum size should be computed as follows. Let the number and size (in bytes) of tuples in relations A and B be represented by Ta, Ba, and Tb, Bb, respectively. The maximum possible output is then (Ta * Tb * (Ba+Bb) / BLKSIZE) blocks (where BLKSIZE is the block size in bytes) rather than 2 * (Ta * Ba / BLKSIZE) * (Tb * Bb / BLKSIZE) blocks, as computed using the previous method. The maximum possible output size computed using the new method is then ((Ba + Bb) / (2 * Ba * Bb)) * BLKSIZE times the value obtained using the old method. This ratio reduces to BLKSIZE / B when Ba = Bb = B. In this particular simulation the output size is 265.4 blocks (19759 tuples of 175 bytes each) which corresponds to 0.1 selectivity factor using the old method and 0.0007 selectivity factor using the new method. In either case, it is clear that the JOIN operation produces a large amount of output which needs to be transferred to the single output processor (either CC or a Pi).

Finally, the simulation shows that a major factor in determining the operation time is the availability of SM modules for the purpose of broadcasting blocks of B. If even a fraction of the module contains output tuples, the SM cannot be used for broadcast, since the module should be completely empty in order to broadcast a full block of B. Thus, long delays are introduced at broadcast times. This problem is solved by increasing the size of each SM module and designating one part of the module for output and the other part for broadcast. With such an arrangement an SM module is always available for broadcast purposes and no delay is incurred in this stage.

The JOIN operation time is sensitive to the output size (i.e., number of times SM is switched to CC) and also to the number of iterations in the nested-loop algorithm. Figure 3.7 illustrates the strong correlation that exists between the number of scans of relation A and the JOIN operation time (in the figure, the term Block Reads is defined as follows: Block Reads = Number of Blocks in relation * Number of Scans of relation). The number of times that relation A is scanned can be reduced by broadcasting more than one block of B at a time. Also, the relative sizes of the output and broadcast areas in the SM's can be made flexible according to the needs of the specific queries. A larger broadcast area will reduce the number of broadcasts of relation B from a node (and, hence, the number of scans of relation A) whereas, a larger output area will reduce the frequency of service required from CC by a specific node.

The effect on operation time of the relative sizes of the broadcast and output areas is shown in Figure 3.8. The

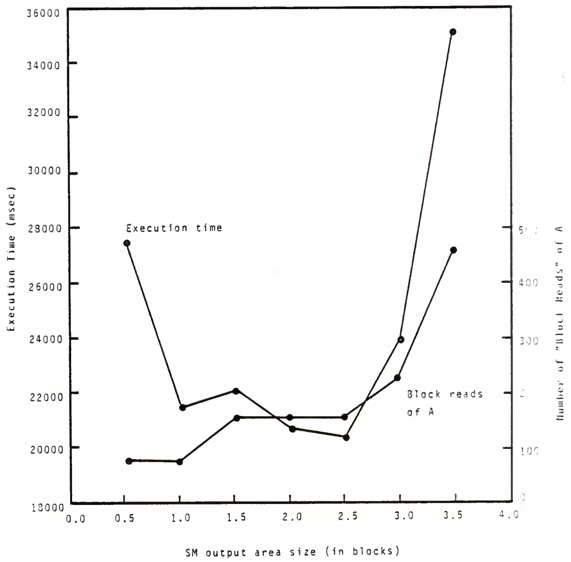


Figure 3.8 Effect of varying broadcast vs. output area sizes in SM. Total SM size = 4.0 blocks; number of processors = 6.

total SM size (output + broadcast) is fixed at 4.0 blocks and the output area size is varied from 0.5 to 3.5 blocks. For an output area size of 0.5 blocks, the operation time is adversely affected by the number of SM transfers required to move the output to CC. Increasing the output area size to 1.0 blocks improves the balance between the output and broadcast areas and the overall operation time decreases. A further increase in the output area reduces the broadcast area and increases the number of cycles required to broadcast all of relation B. Since the entire relation A needs to be scanned for each broadcast cycle, this has the overall effect of increasing the number of block reads of A, which directly affects the operation time. This is shown clearly in Figure 3.8 where increasing the output area from 1.0 to 1.5 blocks and 2.5 to 3.5 blocks increases the block reads on A and, consequently, the JOIN operation time.

The simulation results for the AGG_LIN, AGG_LOG, and AGG_SEQ algorithms for statistical aggregation by categorization are shown in Figure 3.9. The total operation time has been plotted for these algorithms versus the number of categories, C. The results for AGG_LIN and AGG_SEQ are in agreement with the initial timing equations results obtained in [BAR84]. Clearly, AGG_LIN and AGG_LOG perform better than AGG_SEQ for lower number of categories. In AGG_SEQ, the entire relation is broadcast sequentially regardless of the number of categories. The second phase, where the results are collected by the CC, is a small

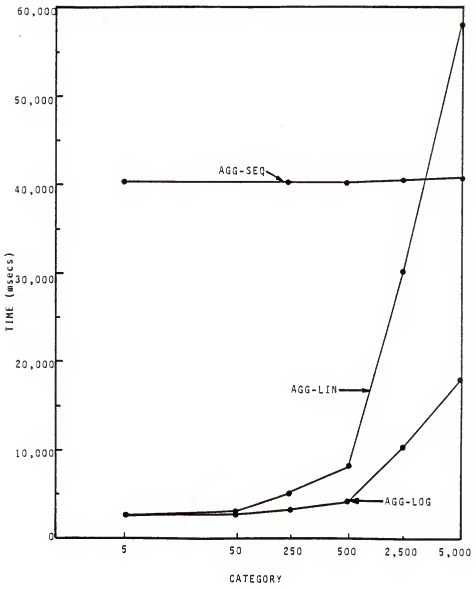


Figure 3.9 Relative performance of the statistical aggregation algorithms vs. number of output categories. Source relation size = 50,000 tuples.

component of the overall operation time. Thus the operation time is always very high and does not increase much with C. The AGG_LOG algorithm performs better than AGG_LIN due to greater parallelism in the second phase, when the global aggregates are being computed. In either case, AGG_LIN and AGG_LOG become worse than AGG_SEQ after a certain value of C is reached. For the values chosen in Figure 3.9, viz. number of processors = 16 and source relation size = 50,000 tuples, AGG_LIN becomes worse than AGG_SEQ after about 3,425 categories and AGG_LOG is worse after 12,660 categories.

A breakdown of the operation time and its components, plotted against the number of processors NP, is given in Figure 3.10 for the AGG_LIN and AGG_LOG algorithms. The first phase, computation of local aggregates, takes exactly the same amount of time in both algorithms and decreases with increasing number of processors. On the other hand, the time taken for global aggregation in the second phase of the operation increases with NP. In AGG_LIN this increase is very rapid with NP due to the sequential nature of operations in the second phase. In AGG_LOG, on the other hand, the increase is very gradual due to the parallelism employed, thus making AGG_LOG a better algorithm than AGG_LIN for the statistical aggregation operation.

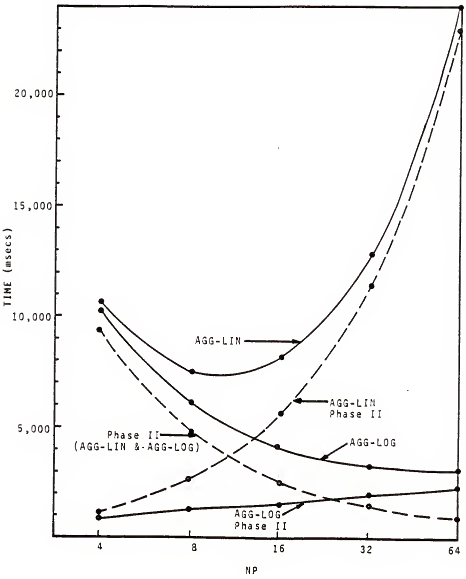


Figure 3.10 Performance of the statistical aggregation algorithms vs. number of processors.

CHAPTER 4
COMPARISON WITH OTHER ARCHITECTURE CLASSES

Based on its architectural features, the SM3 system can be placed in a class between local area networks (LANs) on the one hand and tightly-coupled multiprocessors on the other. The term multicomputer has thus been used to describe the system. This chapter explains the similarities and differences between SM3 and these other architectures.

4.1 SM3 versus Local Area Networks

Just as in any LAN, each node in SM3 is an independent computer capable of supporting "local" operations. In addition, SM3 can support "global" operations which involve two or more processors. The type of operation, "local" or "global", is transparent to the lay user who has a monolithic, "single system" view of the entire multicomputer system. In the "global" mode, SM3 supports asynchronous, SIMD processing. The architectural features of SM3 provide efficient support for this mode of operation, whereas in a LAN, regardless of the speed of the communication medium, the inherent system overheads make the architecture less suitable for such operation.

In order to provide a rough comparison between SM3 and a LAN, the simulation programs were modified to simulate conditions for a LAN. The time taken to transfer a block of data between computers in SM3 is TSWITCH + TMOVE, which accounts for the memory switching time (TSWITCH) and the time taken to read and write data from and to main memory (TMOVE). In a LAN, the same operation takes a total time of TPKT + TMOVE + TXFER, where TPKT is the packet assembly/dissembly overhead, TMOVE is as described above, and TXFER is the time taken to transfer a block of data on the interconnection medium. Unlike SM3, a conventional LAN has no control lines and all interprocessor communication and synchronization is achieved by transferring packets on the network for which a time of TPKT has been assumed. This communication and synchronization overhead increases at least linearly with the number of processors per cluster. Also, since the transmission medium in a LAN is shared by all the network processors, data transfer among Pi's cannot be overlapped with the transfer of data between a Pi and CC. Finally, the architecture of conventional LAN's does not permit dynamic partitioning. Therefore, all processors in a network, regardless of the tasks they are involved in, contend for the same bus thereby further degrading the performance of parallel algorithms.

The SELECT operation is very straightforward and, apart from the additional time required to transfer output to CC, the total operation time in a LAN should be similar to that obtained in SM3. Simulation programs for the JOIN operation in SM3, on the other hand, were modified to obtain corresponding results for a LAN environment. The time taken to transfer data is increased as indicated above. In this operation the processors synchronize after every block of relation B has been broadcast and joined. If bi represents the number of blocks of relation B in the ith node, then in a cluster of P processors, (P-1) $\sum$ ceil(bi) packets need to be transmitted for the purpose of interprocessor synchronization. Figure 4.1 shows the JOIN operation times for SM3 and a LAN. The graph for the LAN is optimistic since the simulation is not very detailed and the assumptions made were in favor of the LAN (e.g., the media access time is not accounted for, overlap between broadcasts of B and transfer of output to CC is permitted, CPU and network operations are assumed to occur concurrently without affecting each other, etc.). As shown in the figure, SM3 consistently performs better than the LAN.

It is reiterated here that the architectural features of SM3 provide certain capabilities which LAN's do not possess. For example, in certain applications large, memory-resident data objects (e.g. matrices) may need to be stored across many processors in order to accomodate them in physical memory. In this case, the SM modules can be used as expanded memory where, depending on the switching of modules, the same address space refers to different data. This type of memory switching and addressing capability is

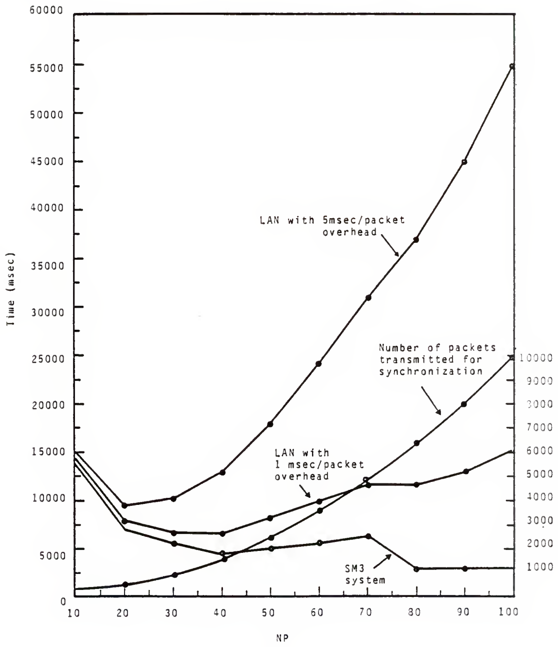


Figure 4.1 JOIN operation execution times in SM3 and a local area network vs. number of processors.

found to be useful especially in certain numeric operations [BAR85]. The dynamic partitioning of the bus is also a very useful feature of SM3 which is not available in LAN´s. For example, the second phase of the AGG_LOG algorithm mentioned earlier makes extensive use of the dynamic partitioning feature. The partitionable bus allows a binary-tree process structure to be mapped efficiently onto the architecture which, as indicated by the simulation, cuts down drastically on the overall operation time.

4.2 SM3 vs Shared Memory Systems

Many variations of a "conventional" shared memory system are in existence. Rather than attempting to simulate all these systems, a qualitative comparison and some discussion of the similarities and differences between SM3 and shared memory systems is provided. A key difference between these two systems is that in shared memory systems a fixed amount of physical memory is mapped to a given address space which is shared by all processors. There is a 1-to-1 mapping between the logical addresses of the shared space and the physical locations of the shared module. In contrast, in SM3 there is a 1-to-n mapping between logical addresses and physical locations. The setting of the SMS switches determines which particular module, i.e. physical location, will be referenced by the given logical address.

Most shared memory systems fit one of the models illustrated in Figures 4.2 to 4.4. The figures show (1) a

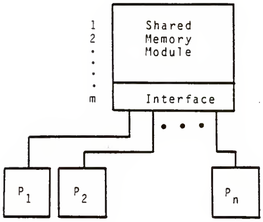


Figure 4.2 A multi-ported memory system

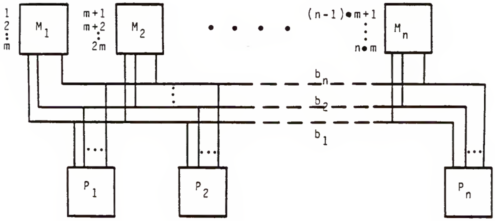


Figure 4.3 A multiple bus shared memory system

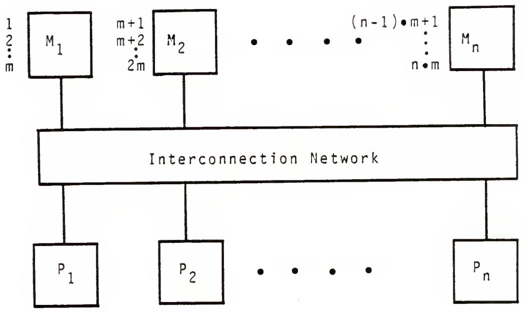


Figure 4.4 A circuit switched shared memory system

multi-ported shared memory module in which each node has a port to memory, (2) many distinct shared modules (mapped to different address spaces) connected to processors via multiple buses so that any processor may access any memory via any one of the buses, and (3) processors and memories interconnected via some switching network, again to allow processors to access any memory. Clearly, all of these schemes are more complex than the switchable memory scheme where all SM´s are connected via a common bus structure and each SM module is interfaced to 3 buses via a simple 1-of-3 multiplexor. Also, it should be noted that the SM modules in SM3 are mainly used to support data transfer among nodes. The memory schemes used in shared memory systems are very useful and efficient for supporting many other kinds of functions but, in general, are too complex for our purposes. Finally, as in the case of the LAN´s, most shared memory schemes do not support dynamic partitioning of the system into independent clusters. In SM3, we permit such partitioning in order to create clusters which are clones of the original network.

CHAPTER 5
MAPPING ALGORITHMS TO ARCHITECTURES

From the discussions in the last three chapters it is clear that performance is a major concern in designing architectures and algorithms. Any new effort in this direction is geared towards providing architectures that improve upon existing systems and towards designing algorithms for these architectures which exploit the available resources to the fullest extent possible. The problem of mapping algorithms to architectures is vast in itself and involves consideration of numerous factors related to the algorithm, architecture, input data, etc. In this chapter we confine ourselves to algorithms for database operations and to database machine architectures. A greater emphasis is placed on multicomputer architectures and on the notion of horizontal parallelism.

The work presented here is a first step towards understanding the problem of mapping database operations to architectures. The material in this chapter provides some guidelines to achieve this mapping and is not expected to be a complete procedure by itself. Given a representation of the algorithm on the one hand and the architecture on the other, the guidelines provide a method for transforming the algorithm representation into a form that can be mapped easily onto the architecture representation. These guidelines can be used to evaluate altenate algorithms for a given architecture or vice versa.

The method for representing algorithms and architectures is based on the contention that a small set of basic structures can adequately represent any algorithm and architecture used for database processing. An intuitive method is suggested for representing and evaluating the algorithms. In fact, we use the structured programing constructs as the building blocks for representation. A given algorithm is transformed, using these constructs, into a form which is suitable for mapping and performance evaluation purposes. This standard representation of the algorithm is used to identify the various forms of parallelism that can be exploited, especially in our context of a multicomputer system.

A basic set of constructs can also be identified for representing the architecture of any database machine. The structure of the underlying system can be transformed into a functionally equivalent structure using these basic constructs. Work in this direction has been done in [ANW85] and we shall base our model of the architecture on this principle. Finally, the problem of mapping a given algorithm onto a multicomputer architecutre along with all the attendant problems of creating, managing, and merging parallel execution streams; evaluating the effect of data distribution on the mapping process; etc, are discussed.

5.1 Architecture Representation

Though a wide variety of database machines (DBM´s) have been proposed and are in existence, no standard classification, model or representation is currently available for such machines. A classification scheme of DBM´s along very general lines has been suggested in [SU80]. This scheme classifies DBM´s based on their special hardware and architectural features like associative memory, cellular-logic devices, etc. However, as the SM3 system has shown, a DBM need not neccesarily be a complex of special-purpose hardware and firmware. A general purpose multicomputer can perform as well as, if not better than, some special purpose DBM´s.

Regardless of whether the underlying machine is a general-purpose multicomputer or a special-purpose machine, we believe that it is possible to identify a set of basic structures which can represent any DBM. Conversely, any DBM can be fully represented (in a functionally equivalent form) using this set of basic structures. This is due to the fact that all DBM´s (or machines used for database processing) attempt to support the basic set of operations or functions required for database processing, whether in hardware, firmware, or software. Thus, a functional representation of such machines using a base set of functions should in fact be possible. The particular implementation scheme (hardware, firmware, or software) only affects the timing associated with each function. Some work in developing a representation for DBM architectures has been done in [ANW85]. The architecture representation should at least (i) identify all the distinct functional units and (ii) indicate the support available for data movement operations in order to be useful for evaluation purposes. This aspect of the mapping process will not be elaborated further except to say that every architecture will be assumed to have a functional representation which provides timing values and other information relevant for mapping.

5.2 Algorithm Representation

The problem of representing algorithms in order to map them to an architecture has been well studied. In our study we are interested only in algorithms used for database processing and in their representation in a form which can be easily mapped to a given architecture representation. The method used for representing algorithms is in itself very general but the discussion in this chapter will pay more attention to the case where the underlying architecture is a multicomputer system. Given such a system, the method of representing algorithms for various operations is first described followed by a study of the different kinds of parallelism that can be exploited for the given algorithm representation.

A "transformational approach" for mapping algorithms to VLSI structures is described in [FOR85]. A transformational system is a three dimensional space where the dimensions are associated with algorithm representation, algorithm model, and architecture specification. The algorithm representation is associated with different forms or levels in which an algorithm can be presented to the transformational system, e.g. as a high-level language code, a set of equations, etc. The algorithm model shows different levels of abstraction used to represent relevant features of the algorithm, e.g. a directed or dataflow graph, a structured flowchart, etc. The architecture specification is associated with the hardware model or level of design in which the hardware is described, e.g. in terms of functional blocks, systolic arrays, etc. The resultant algorithm specification in [FOR85] is at a very low level of detail and is intended to be directly implementable in VLSI. In this study the representation and mapping of algorithms is at a much higher (functional) level of detail.

A common model used for representing the process structure of an algorithm uses nodes and arcs to indicate the processes and their sequence of execution [GAJ85]. Some work has also been done in studying the various aspects of algorithms for MIMD processing and parallel algorithms for multiprocessor/computer systems [LIN81]. Some typical techniques used in modeling algorithms include queueing models, various versions of the Petri net model like timed and stochastic Petri nets [MOL84], and probabilistic models [BAU75]. The modeling approach adopted here, using structured programming (SP) constructs, attempts to represent the macroscopic rather than the microscopic details of the algorithms. Probabilistic and statistical techniques are best suited for this level of representation. The SP constructs can be used to capture more of the functional information without being burdened by low level details. As described below, the SP constructs are, in fact, good tools for understanding the various implications of parallel processing on database operations. The guidelines given below transform an algorithm, in stages, from a high-level specification into a form which can be easily mapped onto architectures. The mapping of the decomposed algorithm to the architecture is "static" in nature. That is, it is done before the algorithm is executed rather than during the process of execution.

5.3 Deriving the Modified SP Flowchart

Figure 5.1 illustrates the sequence of transformations that an algorithm undergoes from a high-level, English language description into a form which can be mapped onto an architecture. The list of transformations is representative of the overall set and is not meant to be exhaustive in nature. The high-level description of the database operation is transformed to an SP flowchart, according to the particular algorithm being used. For example, one could use the nested-loop, sort-merge, or hashing algorithms for the JOIN operation. This flowchart is modified and re-structured in order to remove details which are irrelevant

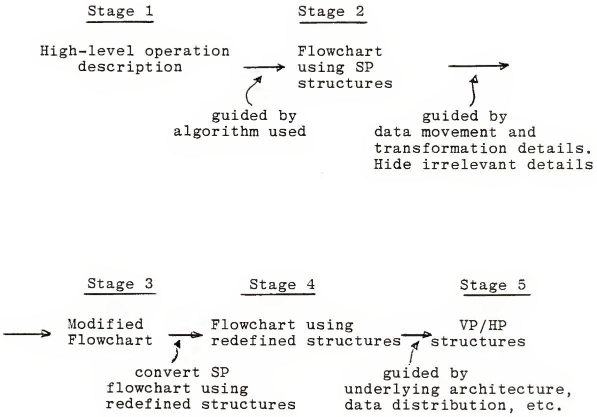

Stage 1
High-level operation description
Stage 2
Flowchart using SP structures
guided by algorithm used
guided by data movement and transformation details. Hide irrelevant details
Stage 3
Modified Flowchart
Stage 4
Flowchart using redefined structures
Stage 5
VP/HP structures
convert SP flowchart using redefined structures
guided by underlying architecture, data distribution, etc.

Figure 5.1 Stages in algorithm representation and transformation

for mapping purposes. The specifics related to volume of data, selectivities, probabilities, etc. are applied to the modified flowchart which is again transformed into a flowchart consisting of SP-type structures whose meaning is redefined as described in Section 5.4. Finally, this representation is used to map the algorithm onto an underlying architecture. This final form of the algorithm can be transformed into various structures representing vertical or horizontal parallelism, as described in Section 5.5.

An algorithm is the "description of a pattern of behavior, expressed in terms of a well-understood, finite, repertoire of named (so-called "primitive") actions of which it is assumed a priori that they can be done (i.e., can be caused to happen)" [DJI76]. It is typically selected on the basis of space and time complexity, cost of execution, ease of implementation, etc. We are interested in developing representations of algorithms for operations such as JOIN, AGGREGATION, SELECT, etc. that are used in processing relational databases. All operations in database processing read some input, perform some computations on the input, and produce some output. This sequence of read-process-output is central to all operations and will be the defined canonical form for describing any algorithm.

The statistical aggregation by categorization operation, described in Section 3.4, will be used as the standard example in order to explain the mapping process.

Figure 5.2(a) shows Stage 1 of the algorithm representation in which the algorithm for the AGGREGATION operation has been described in a high-level, read-process-output form. The projection, categorization, and summarization steps are part of the process portion of the algorithm. Tuples are read one by one and projected on the category and summary attributes. Each tuple is sent to its appropriate category and the corresponding summary value is updated. In the output phase, after the tuples have been read and processed, all the categories are concatenated to obtain the final output. The six original SP structures shown in Figure 5.3, viz. Sequence, If_Then, If_Then_Else, Case, While_Do, and Repeat_Until, are used to construct a detailed flowchart. Figure 5.2(b) shows such a flowchart which corresponds to Stage 2 of the transformation process shown in Figure 5.1. Thus, the transformation from Stage 1 to 2 is guided by the algorithms being used and by the well-established rules of representing algorithms using SP structures.

The detailed flowchart obtained for Stage 2 is transformed to represent an "appropriate" level of detail and, consequently, to arrive at the "modified flowchart" of Stage 3. It is important to consider the level of detail upto which an algorithm should be represented using these SP constructs in Stage 3. Transformation from Stage 2 to 3 is guided by the consideration that the algorithm representation in Stage 3 should be at a level beyond which further decomposition will only add to the details of

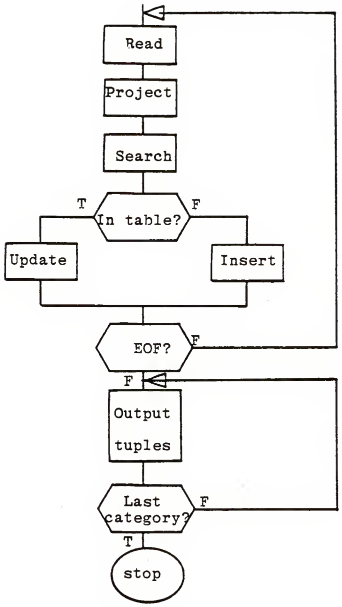
<u>Stage 1</u>

1. Read tuples
2. Project on Attributes
3. Categorize tuples
4. Summarize data
5. Concatenate and output list of categories

(a)

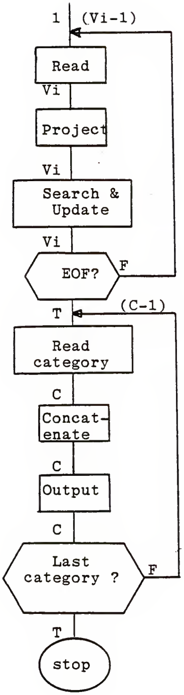
(a)

Figure 5.2 Representation and transformation of the aggregation operation. a) Stage 1; b) Stage 2; c) Stage 3; d) Stage 4.



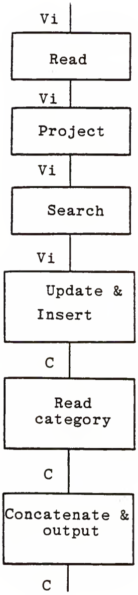
(b)

Figure 5.2--continued



(c)

Figure 5.2--continued



(d)

Figure 5.2--continued

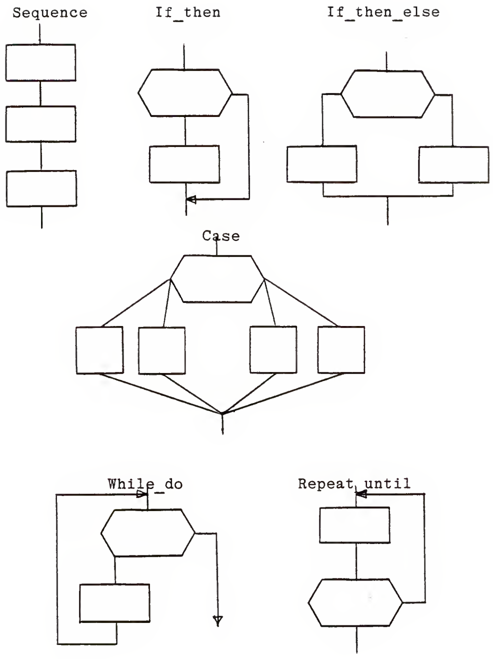


Figure 5.3 The structured programming constructs

execution and has no effect on mapping to an architecture. The key aspects in non-numeric (and even some large-scale numeric) computations are (i) movement and (ii) transformation of data. The steps in the algorithm related to these aspects are shown explicitly while the remaining details can, possibly, be hidden. "Data movement" refers to the I/O and interprocessor transfer of data. "Data transformation" refers to the type of processing: (i) data reordering e.g. sort, (ii) data filtering e.g., select, project, (iii) data aggregation e.g., sum, count, statistical aggregation, and (iv) functional tansformation where the input and output data have some functional relationship. As described later in Section 5.5, these transformations are also important to the mapping process. Thus, any step in the algorithm which is related to data movement and/or transformation needs to be retained. In the transformation from Stage 2 to 3, the decision to "expand" certain steps and "compress" others is guided by the above considerations.

For example, the output phase of the AGGREGATION algorithm in Fig. 5.2(b) is recognized to be a distinct phase of the algorithm. It has its own input-process-output structure involving data movement in both the input and output phases. The data transformation itself is merely a concatenation operation which can be viewed as a reordering transformation. The data movement and transformation features are better captured by decomposing this stage into the following three steps, (i) read, (ii) concatenate, and (iii) output as shown in Fig 5.2(c). At the same time, the search and update functions do not involve any data movement or transformations between the two steps. Thus, following the guidelines stated above, they have been combined into a single step. Actually, the search step itself hides details related to the specific implementation (e.g., maintenance of a sorted tree structure, handling of exception condition, etc.) which are not relevant for mapping. Thus, the algorithm representation is essentially stripped of all inconsequential control details and left with only that information which reflects the overall function of the algorithm and only that control which is relevant for mapping purposes.

In sum, the transformation from Stage 2 to 3 decomposes some algorithm steps in order to capture all the data movement and transformation information and recombines others to hide the control and processing information which are irrelevant to the mapping process. The structures in the algorithm representation in Stage 3 have a distinct mapping onto the structures in the architecture representation. For example, suppose an algorithm has a selection step followed by a tuple reformatting step. In a conventional, single-CPU system both these steps are mapped to the CPU whereas, in a system with a filter processor, the select step can be mapped to the filter and the reformatting step to the CPU. As the level of the architecture representation changes, e.g. from a multicomputer system level, to a single-node level, to a chip level, etc., the algorithm decomposition and representation will also change accordingly, thereby allowing the general method outlined here to be used for any level of architectural detail.

5.4 Further Transformations of the Algorithm

Following Stage 3 described above, the objective in transforming an algorithm to Stage 4 of Fig. 5.1(d) is to provide a representation which has a structure, along with all the relevant details, which can be mapped onto an architecture.

As mentioned before, the modified flowchart of Stage 3 is transformed into a redefined flowchart in Stage 4 using a set of redefined SP structures. This flowchart retains the essential control flow and all the data flow information from the flowchart for the original algorithm. The algorithm representation is now in a form which can be mapped onto the architecture. The boundary between algorithm and architecture is now crossed and the information related to data volumes and estimated processing time per step per unit of data can be encoded in the flowchart. The meaning ascribed to some of the redefined SP structures will now be explained.

The redefined Sequence structure is shown in Figure 5.4. Each box may contain many instructions which have been grouped together. Operations in different boxes cannot be

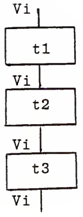


Figure 5.4
Redefined
Sequence

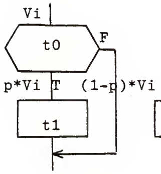


Figure 5.5
Redefined
If_Then

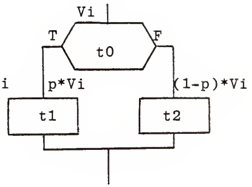


Figure 5.6
Redefined
If_Then_Else

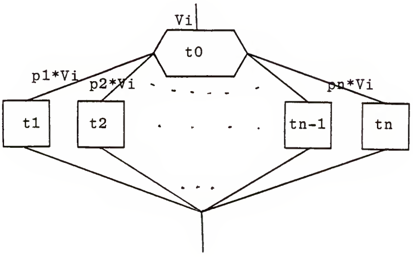


Figure 5.7 Redefined Case

combined. Each box can be mapped to a different architectural structure. For example, one box might map to I/O and the other to CPU or both the boxes may map to two different CPU´s, etc. Each box is assigned a time value (t1, t2, t3, etc.) depending upon the instructions in each box and the characteristics of the underlying architecture. Note that the units used for the volume of data and the processing times should be consistent. The unit of time is typically specified for a certain unit of data which shall be referred to as the "granularity of data". For example, the timing values may be specified in, say seconds per byte, per tuple, per block, etc. The granularity of data is dependent upon the level of detail at which the architecture is represented.

The If_Then structure is shown in Figure 5.5. This structure is redefined to represent a filter mechanism. A volume Vi of data flows into this structure and only the fraction p*Vi of it "satisfies" some condition and is, therefore, subjected to further processing in the box. The quantity p represents the probability that a particular input data satifies the given condition. The fraction (1-p)*Vi of data bypasses the box. The time factor associated with this structure is (t0 + p*t1)*Vi. If the underlying architecture had a filter processor then this entire structure would be mapped to that processor and the time values for t0 and t1 would depend upon the characteristics of this processor.

The If_Then_Else structure is shown in Figure 5.6. The volume of the input data is again Vi and p*Vi amount of data is processed by the box on the left whereas (1-p)*Vi amount of data is processed by the box on the right. The overall time taken is (t0 + p*t1 + (1-p)*t2)*Vi, for this structure. The Case structure shown in Figure 5.7, is a generalization of the If_Then_Else structure. It has a single input with Vi volume of data and multiple outputs with the ith output branch having a volume pi*Vi of data and the box on the ith branch taking ti amount of time to process each unit of data. The overall time for this structure is Vi * ($\sum$ pi*ti + t0). Note that this is a general structure which can be used to represent a given data distribution. The decision box inspects the input data and routes it to the appropriate output branch according to the data distribution criterion supplied to it.

In order to have a complete representation of the algorithm which is also useful from a performance evaluation point of view, the information related to the volumes of input and output data, selectivity factor, branching probability, etc. needs to captured. The timing value assigned to each SP construct in the representation depends on the underlying architecture. The important factors in mapping from Stage 3 to 4 then, are (i) the need to preserve information related to data volumes and (ii) the understanding that the SP-type structures used in the representation in Stage 4 have been redefined. Returning to the AGGREGATION example, Fig. 5.2(d) shows the mapping from Stage 3 to 4. The resultant flowchart in this case has a Sequence structure and the input and output data volumes have been specified for each box.

Thus the process by which a given operation is transformed and expressed in a form that is convenient for our mapping purposes is as follows. First, the given operation is expressed in a very high level form using the canonical input-process-output formulation. Second, a particular algorithm for this operation is expressed using a "conventional" SP flowchart and, third, this flowchart is modified to capture all data movement and transformation details and delete certain inconsequential steps, thereby obtaining a "modified" flowchart. At this stage the information related to data volumes, selectivities, probabilities, etc. is applied to the modified flowchart. Fourth, this modified flowchart is transformed using redefined SP constructs such that it retains some of the control and all of the data flow information of the algorithm. Finally, this representation is amenable to mapping onto architectures. The various mapping schemes that are possible are determined by the underlying architecture. The next section briefly touches upon the assumptions made regarding the architecture representation. The mapping schemes attempt to provide faster and more efficient execution by exploiting vertical and horizontal parallelism in the algorithm. These two kinds of parallelism are discussed in Section 5.5.

5.5 Vertical and Horizontal Parallelism

The representation of an algorithm obtained by following the steps outlined above is used for identifying the various possible mappings of the algorithm to the architecture with a view towards minimizing operation time. We intend to minimize the operation time by using parallel/concurrent processing techniques. The amount of parallelism that can be extracted depends on the algorithm as well as the architecture. Our guidelines for representing algorithms, as given above, emphasize exploiting the parallelism present in the process structure of the algorithm. The amount of parallelism achieved is therefore determined to a great extent by the process structure of the algorithm and by the underlying architecture. The type of operation itself only influences the mechanisms that would be used to control i.e. how to fork & join or create & merge, etc., the execution of the parallel parts of an algorithm, once they have been identified.

This section describes the two main types of so-called large-grain parallelism that we wish to exploit viz., vertical and horizontal parallelism. In the algorithm representation described in Section 5.1, the steps shown in-line by separate boxes can be executed as concurrent processes if they can be mapped to the available architecture. This kind of assignment is referred to as vertical parallelism (VP). This method of processing has also been termed as "vertical concurrency" [GUO85], "macroscopic dataflow" [MAP81], and "macropipelining" [HWA84]. On the other hand, a particular box may be split horizontally into many boxes indicating many different physical processors. This type of assignment is called horizontal parallelism (HP) and it introduces additional stages for splitting and combining the control and data aspects of the algorithm.

5.5.1 Vertical Parallelism

In vertical parallelism (VP), the distinct steps of an algorithm are processed concurrently by many processors. Parallelism is achieved by the overlapping of these steps and the extent of overlapping is determined by the structure of the algorithm. The VP technique is similar to the data flow methods except that it attempts to exploit the parallelism for any underlying machine, especially, multicomputer systems. Individual boxes or sub-structures of the flowchart are mapped onto distinct entities in the architecture. In the case of SM3, each box or sub-structure can be assigned to a pool of processors, i.e. a cluster. A similar technique is used in the MIDAS system [MAP81]. In Figure 5.8(a) the flowchart consists of three boxes representing input, process, and output. Each of these

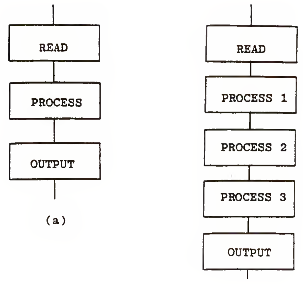


Figure 5.8 Candidates for vertical parallelism.
a) Case 1; b) Case 2

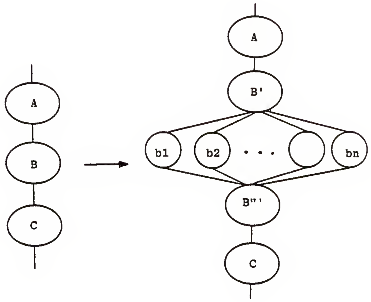


Figure 5.9 Mapping for horizontal parallelism

boxes can be mapped to, say an I/O processor, a CPU, and a third processor which handles data output. Another example flowchart is shown in Fig. 5.8(b) where after input, the processing part is split into three distinct phases as indicated by the three boxes. Each of these boxes can be mapped to different physical processors in order to provide vertical parallelism. Thus, VP attempts to overlap the execution of the different parts of an algorithm by assigning these parts to different processors. At any instant in time, the various parts are active and operating on different instances of data. Our problem of mapping VP processes to architectures is similar to the work in [HON85], where data flow graphs are derived for algorithms on a specific (hypercube) architecture. The scheme used there ensures that concurrent tasks run on different processors.

Vertical parallelism is possible only if the flowchart lends itself to such mapping and if the architecture can support the mapping. In the case of mapping separate functions (e.g., I/O, computation, output, etc.) to processors, the underlying hardware should be able to support the functions assigned to it. When similar functions (say, computation alone) are mapped to different processors, the architecture should support appropriate communication between such processors. Thus, in order for VP to work, one needs the ability to map VP activities to distinct entities in the architecture along with some implicit or explicit facility to transfer data between these entities. As far as the data used as input in the algorithm is concerned no special consideration is required. The data flow remains the same as indicated in the original flowchart.

5.5.2 Horizontal Parallelism

In horizontal parallelism, the data to be processed are shared among many processors. In terms of the algorithm representation, a single box or sub-structure is assigned to many processors and each processor acts concurrently on a different segment of data. The mapping of each box or sub-structure is considered in isolation from the rest of the algorithm. Thus, the process of mapping HP activities to architectures does not alter the overall flow of control. A single step of a flowchart is typically transformed in the manner shown in Figure 5.9, with some overhead associated with creating and merging parallel paths. Any arbitrary step of an algorithm can, possibly, be subject to HP. It is important to consider how the parallel paths are created and merged in HP, since this has a direct effect on the time and cost of the algorithm.

In order to create parallel paths in HP, the data need to be available at the processors assigned to the operation. Data can be distributed, for example, using a simple division rule. If the amount of input data is R units and the number of processors is N then the ith processor is assigned the ith block of R/N units of data. Alternatively, a distribution scheme can use some aspect of the data itself to determine the distribution. Hashing techniques use the value of some of the fields (or attributes) of the data for this purpose. The advantage of a hashing scheme is that it can potentially reduce the time and cost of the HP step, as discussed below. At the same time, it is possible that the data value distribution is such that a hashing scheme loads one parallel path in the HP step with more data than the others thus resulting in load imbalance and consequent degradation in performance. A solution is to derive data distribution schemes which can automatically adjust loads across parallel paths. A special case arises when the data distribution is given. In this case the data can be either re-distributed according to one of the above schemes or used as they are. Regardless of which method is used, the most important information, from the performance evaluation point of view, is the a priori knowledge related to the distribution of data among the parallel paths. The cost and complexity of the HP scheme, especially the scheme used for merging the parallel paths, depends heavily on whether the data distribution is arbitrary in nature or whether the distribution is favorable from the point of view of the algorithm.

From a control point of view, the data transformations carried out by the step(s) involved in the HP have an impact on the mapping of the algorithm. The type of transformations used and the data distribution determines the relationship among the data across parallel paths. For example, for the SELECT operation the data transformation involves a filter operation. The final result is merely a concatenation of the results from each parallel path regardless of the data distribution. On the other hand, the PROJECT operation which consists of a local elimination of duplicates followed by a global elimination, requires data from each parallel path to be compared with the data in every other parallel path. Such a comparison would not be required if it was known a priori that the given data distribution is such that duplicates do not exist across processors.

It is important to consider how data from the parallel paths of an HP algorithm are finally combined. Keeping in mind the various data distribution schemes discussed above, we can list some standard ways of combining data from HP streams. The first case is the simplest and can be viewed as the concatenation of results, where results from each path are collected as they become available and directed to the next step. This is typical of a SELECT operation in which the sequence of output is not significant. The second method for combining data from the parallel paths is to merge or combine data in order to resequence or recompute some values and produce a single, final output. This combination scheme, also referred to as "recursive" or "iterative compute-combine" [HWA84], is characteristic of operations like SORT and AGGREGATION. Adopting a binary tree process structure will obviously make this process faster, provided the architecture can support it. In the third type of data combination scheme data from each path are sent to every other path in the HP task. This is typical of the PROJECT operation discussed above.

Another method of exploiting horizontal parallelism in an algorithm is to divide the control aspect of a single step across the parallel paths as opposed to dividing the data. An example is the splitting of the conditional expression of a SELECT operation expressed either in the conjunctive or disjunctive normal form (CNF or DNF). In this case each path executes a part of the original operation as opposed to every path executing the same instructions. This kind of parallelism may also be viewed as an MISD process structure. Two possibilities exist for synchronizing parallel paths. In the conjunctive case, all paths should reach completion before synchronization and we call this the "global-AND" situation. The final output is the set intersection of all the inputs. In the disjunctive case, the completion of any one path is sufficient for synchronization and we call this the "global-OR" situation. The final output in this case is the set union of all inputs.

5.6 Representation of Some Sample Algorithms

The algorithm representation and transformation for the SELECT and PROJECT operations are presented here as examples. Stages 1 to 3 of representation for the SELECT operation are shown in Figures 5.10(a)--(c). Figure 5.10(c) can also be used to represent the algorithm in Stage 4, since the transformation from Stage 3 to 4 does not significantly change the representation in this case. The algorithm used for the SELECT operation is quite straightforward. The if_then structure shown in Figure 5.10(b) can be directly mapped to an available filter processor. The process portion of the algorithm can be subject to HP in which case, as described above, the final results need to be concatenated before being output.

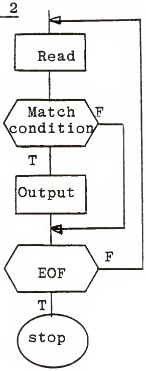
The various stages for the PROJECT operation are shown in Figures 5.11(a)--(c). As shown in the figures, only D tuples are output from a total of Vi input tuples. Thus, the remaining (Vi-D) tuples are duplicates. Again, the process step can be subject to HP. If the data are distributed on the basis of the projection attribute then a simple concatenation can be used to combine the parallel paths. If an arbitrary distribution scheme is used then the third combination scheme mentioned above should be used.

Stage 1

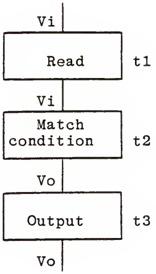
1. Get tuples
2. Match against condition
3. Output all tuples that satisfy condition

(a)

Stage 2



(b)



(c)

Figure 5.10 Algorithm representation for the SELECT operation. a) Stage 1; b) Stage 2; c) Stage 3.

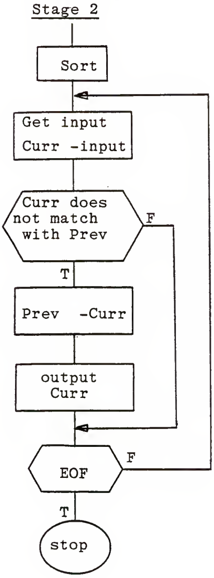
Stage 1

1. Get tuple

2. For this tuple remove all other tuples with the same information

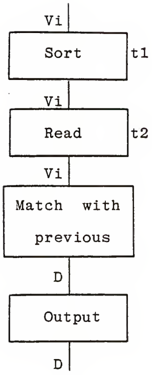
3. Output this tuple

(a)



(b)

Figure 5.11 Algorithm representation for the PROJECT operation. a) Stage 1 b) Stage 2; c) Stage 3.



(c)

Figure 5.11--continued

CHAPTER 6
SUMMARY, CONTRIBUTION AND FUTURE WORK

The architecture of a dynamically partitionable multicomputer system with switchable memory was presented as a viable alternative for database processing. The multicomputer system employs the architectural concepts of switchable memory, global control lines, and dynamic partitioning of a common-bus network in order to support efficient processing of relational database operations. A detailed description of the design and functioning of the hardware was given and the results obtained from discrete-event simulations of some common database operations were discussed. A set of guidelines were also provided to map database algorithms to architectures. The guidelines are meant to help in restructuring algorithms expressed in the form of structured flowcharts into a form which allows the exploitation of the available vertical and/or horizontal parallelism for the algorithm.

The contribution of the work presented here is that it has provided a scheme for interconnecting multiple computers for supporting database processing. The architectural features introduced aid in efficient processing of database operations. It has been shown that a multicomputer system with a suitably designed system architecture can support database processing more efficiently than most special purpose database machines. In addition, the general purpose nature of such an architecture allows a variety of algorithms to be implemented on it. Thus, one can choose from a wide range of algorithms, selecting the most efficient one under the given circumstances. This work also provides a methodology for representing and mapping algorithms for database processing onto underlying database machine architectures with a view towards evaluating alternative algorithms.

Many interesting and important issues still remain open. These include issues related to data distribution, query processing, processor allocation, and flexible or dynamic algorithm selection in multicomputer systems. Recently some preliminary work was done in studying the use of SM3 for large-scale numeric computations [BAR85]. Further research needs to be done in this area also. We shall briefly explain each of the above issues.

6.1 Data Distribution

Under data distribution one needs to study the problem of optimal distribution of data across processors such that the parallel processing capabilites of the machine are utilized to the maximum in executing individual operations, individual queries, and multiple queries. Most of the existing work in data distribution is oriented towards systems in which the communication costs are high.

Parallelism is measured in terms of the number of nodes executing different operations/queries at any given time. There is a strong motivation to reduce "non-local" accesses in such systems since they incur high communication costs and, also, tie up more than one node for the same operation/query. On the contrary, in a multicomputer system one gains performance by utilizing many processors for a single operation and the concern in SM3 is the distribution of data for maximizing parallelism rather than minimizing communication cost.

One might consider using the semantic knowledge of the data and the a priori knowledge related to query patterns in deciding the data distribution scheme. It should be noted that in SM3 there is a direct mapping between the logical connection between relations and the physical proximity of processors containing the relations. Relations that are referenced together (say, in a JOIN operation) should be physically close together (preferably on the same set of processors) in order to minimize cluster sizes and maximize concurrency.

6.2 Query Processing and Processor Allocation

The SM3 system should support as many active queries in the system as possible, i.e. it should support a high degree of inter-query concurrency. At the same time, response times should be reduced by processing the operations in a query in parallel, i.e. by employing intra-query concurrency and parallel processing of database operations. In order to maximize inter-query concurrency, individual queries should map to disjoint sets of processors. For each query, in order to maximize intra-query concurrency, individual sub-trees of the query tree should again map to disjoint sets of processors. In the limit each unary operation at the leaf node of a query tree should map to a distinct set of processors. If all the relations involved in a query are stored "close" to each other then the query can be processed by a small cluster of processors each of which has a segment of one of the relations and is thus, actively involved in the query execution. If the relations are dispersed across non-adjacent processors then larger clusters, involving idle processors, need to be formed. Base relations could be in a dispersed form due to the initial data distribution and temporary relations could be dispersed as a result of the previous step of processing.

A formal description of this process will help in defining and understanding this problem. An example is shown using the SELECT operation. Let $\{i\}$, i = 1, N be the index set of processors. Each processor has Ri blocks (or tuples) of the relation R and $\sum Ri = |R|$ is the total relation size (i.e., no redundancy in data).

The set of (i, Ri) tuples, $\{(i, Ri)\}$, i = 1, N refers to the set of processors (cardinality N) which fully contain relation R, with the ith processor containing Ri blocks of

R. The SELECT operation, denoted S, can have the following effects:

1. $S(\{(i, Ri)\}), i = 1, N \rightarrow \{(i, Si)\}, i = 1, N$. i.e., each of the Ri blocks have been reduced to Si blocks, but each of the N processors has some blocks of output (i.e. $Si > 0, i = 1, N$).
2. $S(\{(i, Ri)\}), i = 1, N \rightarrow \{(i, Si)\}, i = 1, M, M<N$. i.e., each of the M processors have Si blocks of output ($Si > 0, i = 1, M$) and N-M processors have zero blocks of output and have, therefore, dropped out.

Query processing is affected by two factors (i) size of each Si and (ii) value of N-M. Ideally, we would like to distribute the data such that $N = M$ and $Si = Sj$ for all $i, j = 1, N$. It may, for example, be cost effective to dynamically modify the query tree and introduce a redistribution operation in order to execute every operation with an optimal number of processors.

6.3 Large Scale Numeric Computation

Some preliminary work has been done in studying the performance of certain matrix operations (specifically, matrix multiplication) for very large matrices in the SM3 system [BAR85]. Numeric computations involving large inputs, e.g. a 10,000 x 10,000 matrix of 64-bit floating point numbers, need to contend with not only the large computational component but also the significant I/O component of the operation. A multicomputer system like SM3 can provide a good balance between CPU and I/O power to support such operations. More work needs to be done to study operations beyond the multiplication operation in SM3. The architecture may undergo a modification as a consequence of this study.

REFERENCES

AHU82 Ahuja, S.R., and Asthana, A. A Multi-Microprocessor Architecture with Hardware Support for Communication and Scheduling, Proc. of Architectural Support for Programming Languages and Operating Systems, ACM/SIGARCH, March 1982, pp. 205-209.

ANW85 Anwar, A. ME theis, Department of Electrical Engineering, University of Florida, Gainesville, Fl. In preparation, 1985.

ARD81 Arden, B.W., and Ginosar, R. MP/C: A Multiprocessor/Computer Architecture, Proc. ACM/SIGARCH 8th Intl. Conf. on Computer Architecture, 1981, pp. 3-19.

BAN79 Banerjee, J., Hsiao, D.K., and Kannan, K. DBC: A Database Computer for Very Large Databases, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 414-429.

BAN80 Bancilhon, F. and Scholl, M. Design of a Backend Processor for a Data Base Machine, Proc. ACM/SIGMOD Intl. Conf. on Management of Data, May 1980, pp. 93-93g.

BAR83 Baru, C.K. The Shared Main Memory Modules (SM3) System: Architecture and Performance Evaluation, Master´s thesis, Department of Electrical Engineering, University of Florida, 1983.

BAR84 Baru, C.K., and Su, S.Y.W. Performance of Statistical Aggregation Operations in the SM3 System, Proc. ACM/SIGMOD Intl. Conf. on Management of Data, June 1984, pp. 77-89.

BAR85 Baru, C.K., Thakore, A., and Su, S.Y.W. Matrix Multiplication On a Multicomputer System With Switchable Main Memory Modules, to be presented at the IEEE First Intl. Conf. on Supercomputing Systems, December 16-20, 1985, St. Petersburg, Florida.

BAT77	Batcher, K.E. STARAN Series E, Proc. 1977 Intl. Conf. on Parallel Processing, August 1977, pp. 140-143.
BAU75	Bauer, F.L., Software Engineering, Springer-Verlag, Germany, 1975.
BOR80	Boral, H., and DeWitt, D.J. Design Considerations for Data-Flow Database Machines, Proc. ACM/SIGMOD Intl. Conf. on Management of Data, May 1980, pp. 94-104.
CAN74	Canady, R.H., Harrison, R.H., Ivie, E.L., Ryder, J.L., and Wehr, L.A., A Back-End Computer for Database Management, Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 575-582.
CHA81	Chamberlin, D., Astrahan, M.M., King, W.F., Lorie, R.A., Mehl, J.W., Price, T.G., Schkolnick, M., Griffiths Selinger, P., Slutz, D.R., Wade, B.W., and Yost, R.A. Support for Repetitive Transactions and Ad Hoc Queries in System R, ACM Transactions on Database Systems, Vol. 6, No. 7, March 1981.
DEW79	DeWitt, D.J. DIRECT: A Multiprocessor Organization for Supporting Relational Database Management Systems, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 395-406.
DEW81	DeWitt, D.J., and Hawthorn, P.B. A Performance Evaluation of Database Machine Architectures, VLDB-81, Proc. 7th Intl. Conf. on Very Large Data Bases, France, September 1981, pp. 199-213.
DJI76	Djikstra, E. The Art of Programming, Computer Society of India, Bombay, 1976.
EPS80	Epstein, R., and Hawthorn, P. Design Decisions for the Intelligent Database Machine, AFIPS Conference Proceedings, NCC, 1980, pp. 237-241.
FEI84	Fei, T., Baru, C.K., and Su, S.Y.W. SM3: A Dynamically Partitionable Multicomputer System with Switchable Main Memory Modules, Proc. IEEE/COMDEC, Intl. Conf. on Computer Data Engineering, April 1984, pp. 42-49.
FOR85	Fortes, J.A.B., Fu, K.S., and Wah, B.W. Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays, IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, Tampa, Fl, 1985.

GAJ85	Gajski, D.D. and Peir, J.K. Essential Issues in Multiprocessor Systems, IEEE Computer Vol. 18, No. 6, pp. 9-27.
GOR80	Gorsline, G. Computer Organization: Hardware /Software, Prentice-Hall, Englewood Cliffs, N.J., 1980.
HON85	Hong, Y.C., Payne, T.H., and Oddson, J.K. Efficient Computation of Dataflow Graphs in a Hypercube Architecture, Dept. of Math. & Computer Science, Univ. of California, Riverside, Ca.
HWA84	Hwang, K. and Briggs, F.A. Computer Architecture and Parallel Processing, McGraw-Hill, N.Y., 1984.
IEE79a	IEEE Transactions on Computers, Special Issue on Databse Machines, Vol. C-28, No. 6, June 1979.
IEE79b	IEEE Computer, Special Issue on Data Base Machines, Vol. 12, No. 3, March 1979.
IEE84	The Genesis of a Database Computer, IEEE Computer, Vol. 17, No. 11, November 1984, pp. 42-56.
JOH82	Johnson, R. and Thompson, W. A Database Machine Architecture for Performing Aggregations, Technical Report, Univ. of California, Lawrence Berkeley Labs, UCRL-87419, June 1982.
KAR78	Kartashev, S.I., and Kartashev, S.P. Dynamic Architectures: Problems and Solutions, IEEE Computer, Vol. C-27, No. 7, pp. 26-41, July 1978.
KIM80	Kim, W. A New Way to Compute the Product and Join of Relations, Proc. ACM/SIGMOD Intl. Conf. on Management of Data, May 1980, pp. 179-187.
KUN80	Kung, H.T., and Lehman, P.L. Systolic (VLSI) Arrays for Relational Database Operations, Proc. ACM/SIGMOD Intl. Conf. on Management of Data, May 1980, pp. 105-116.
LIN76	Lin, C.S., Smith, D.C.P., and Smith, J.M. The Design of a Rotating Associative Memory for Relational Data Base Applications, ACM Transactions on Database Systems, Vol. 1, No. 1, 1976, pp. 53-65.
LIN81	Lint, B., and Agerwala, T. Communication Issues in the Design and Analysis of Parallel Algorithms, IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, March 1981, pp. 174-188.

MAP81 Maples, C., Rathbun, W., Weaver, D., and Meng, J. The Design of MIDAS: A Modular Interactive Data Analysis System, IEEE Transactions on Nuclear Science, Vol. NS-28, No. 5, October 1981, pp. 3746-3753.

MIS82 Missikoff, M., and Terranova, M. An Overview of the Project DBMAC for a Relational Database Machine, Technical Report, IASI-CNR, Rome, Italy, 1982.

NIC80 Nickens D.O., Genduso, T.B., and Su, S.Y.W. The Architecture and Hardware Implementation of a Prototype MICRONET, Proc. 5th Intl. Conf. on Local Computer Networks, October 1980.

OZK75 Ozkarahan, E.A., Schuster, S.A., and Smith, K.C. RAP: An Associative Processor for Data Base Management, AFIPS Conference Proceedings, NCC, 1975, pp. 370-387.

RAS85 Raschid, L., Fei, T., Lam, H., and Su, S.Y.W. A Special Function Unit for Sorting and Sort-based Database Operations, submitted for publication, 1985.

SCH79 Schuster, S.A., Nguyen, H.B., Ozkarahan, E.A., and Smith, K.C. RAP.2: An Associative Processor for Databases and its Applications, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 446-458.

SHA79 Shaw, D.E. A Heirarchical Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives, Technical Report STAN-CS-79-778, Dept. of Computer Science, Stanford University, October 1979.

SHA81 Shaw, D.E., Salvatore, J.S., Hussein, I., Hillyer, B., Wiederhold, G., and Andrews, J.A. The NON-VON Database Machine: A Brief Overview, IEEE Database Engineering, Vol. 4, No. 2, December 1981.

SU78 Su, S.Y.W., Lupkiewicz, S., Lee, C.J., Lo, D.H., and Doty, K.L. MICRONET: A Microcomputer Network System for Managing Distributed Relational Databases, VLDB-78 Proc. 4th Intl. Conf. on Very Large Data Bases, Berlin, West Germany, September 13-15, 1978.

SU79 Su, S.Y.W., Nguyen, L.H., Emam, A., and Lipovski, G.J. The Architectural Features and Implementation Techniques of the Multicell CASSM, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 430-445.

SU80 Su, S.Y.W., Chang, H., Copeland, G., Fisher, P., Lowenthal, E., and Schuster, S. Database machines and some issues on DBMS Standards, AFIPS Conference Proceedings, NCC, 1980, pp. 191-208.

SU83 Su, S.Y.W. A Microcomputer Network System for Distributed Relational Databases: Design, Implementation, and Analysis, Journal of Telecommunication Networks, Vol. 3, No. 2, Academic Press, Fall 1983.

SU84 Su, S.Y.W., and Baru, C.K. Dynamically Partitionable Multicomputers with Switchable Memory, Journal of Parallel and Distributed Computing, Vol. 1, No. 2, Academic Press, November 1984, pp. 152-184.

SWA77 Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm*--A Modular Multi-Microprocessor, AFIPS Conference Proceedings, NCC, 1977, pp. 637-644.

BIOGRAPHICAL SKETCH

Chaitanya K. Baru was born on the 31st of January, 1958, in Hyderabad, India. He obtained his bachelor's degree in electrical engineering in 1979 from the Indian Institute of Technology, Madras, and entered the "real" world by joining the Tata Engineering and Locomotive Company, Ltd., Pune, India. In 1981 he found that he had had enough of the "real" world and decided to enroll as a graduate student in the Electrical Engineering Department at the University of Florida. He received the master's degree in April, 1983 after which he liked the "unreal" world so much that he stayed on for a Ph.D. After his Ph.D. he expects to go north to a place in the Arctic called Michigan.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Dr. Stanley Y. W. Su, Chairman
Professor of Electrical Engineering and Computer Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Staudhammer

Dr. John Staudhammer
Professor of Electrical Engineering and Computer Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Dr. Yuan Chieh Chow
Associate Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Herman Lam

Dr. Herman Lam
Associate Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

S B Navathe

Dr. Shamkant B. Navathe
Associate Professor of
Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1985

Hubert A. Bevis

Dean, College of Engineering

Dean, Graduate School